LA PRATICA DEL COMMODORE 64

3. linguaggio macchina e assembler del 6502

daniel - jean david

La pratica
del COMMODORE 64

3. Linguaggio macchina
e assembler del 6502

DELLO STESSO EDITORE

Volumi pubblicati

D. A. Lien - Dizionario del Basic
J. Boisgontier - Il Basic per tutti
A. Pinaud - CP/M passo dopo passo
D.-J. David - La scoperta del Commodore 64
D.-J. David - La pratica del Commodore 64
J. Deconchat - 102 programmi per Commodore 64
J. Boisgontier - Commodore 64: metodi pratici
J. Boisgontier, S. Brebion, G. Foucault - Il Commodore 64 per tutti
X. Linant de Bellefonds - La pratica dello ZX Spectrum - Vol. 1
M. Henrot - La pratica dello ZX Spectrum - Vol. 2
J.-F. Séhan - Chiavi per lo ZX Spectrum
J. Lévy - Esercizi per lo ZX Spectrum
J.-F. Séhan - Alla ribalta: lo ZX Spectrum
C. Galais - Vademecum per Applesoft
J. Boisgontier - L'Apple e i suoi files
B. De Merly - Guida per l'Apple - Vol. 1
B. De Merly - Guida per l'Apple - Vol. 2
B. De Merly - Guida per l'Apple - Vol. 3
F. Lévy - Esercizi per l'Apple II, II plus, IIe, IIc
J. Boisgontier - 36 programmi per Apple IIe, II plus, IIc
N. Bréaud-Pouliquen - La pratica dell'Apple II - 1. Periferiche e gestione dei file
J.-P. Blanger - Modelli di espressione grafica
A. Pinaud - Programmare in Forth

Volumi di prossima pubblicazione

C. Bardon, B. De Merly - Giochi sul Philips C7420 Videopac +
J. Deconchat - 102 programmi per ZX Spectrum e ZX 81
J. Deconchat, V. Grandis - 102 programmi per il Philips C7420 Videopac +
J.P. Richard - La scoperta del PC 1500
M. Henrot - J. Boisgontier - Lo ZX Spectrum per tutti
J.-F. Séhan - Alla ribalta: il Commodore 64. 30 programmi in Basic
A. Pinaud - MS DOS passo dopo passo
M. Thibault - Chiavi per il MULTIPLAN
C. Nowakhwski - Programmare in C
N. Bréaud-Pouliquen - La pratica dell'Apple II - Vol. 2

DANIEL-JEAN DAVID

La pratica del COMMODORE 64

3. Linguaggio macchina e assembler del 6502

Edizione italiana a cura di
FRANCO POTENZA

Traduzione di
MARCO SPIZZI

Editsi - Editoriale per le scienze informatiche - S.r.l.
MILANO 1985

Daniel-Jean David *insegna informatica gestionale all'Università di Parigi 1 Pantheon Sorbona.*

Insegna tra l'altro l'utilizzazione dei microprocessori all'ENSAM di Parigi.

Il campo di ricerca va dalla grafica computerizzata alle tecniche di interfacciamento di microprocessori ai sistemi multimicroprocessori.

Specialista del microprocessore 6502, ha tenuto, a Parigi, numerosi seminari sui microprocessori, il KIM, il SYM e il PET/CBM. È direttore de "La Commode", rivista trimestrale interamente dedicata ai computer Commodore.

Titolo originale dell'opera
L'ASSEMBLEUR DU COMMODORE 64
Pratique du C.64 - volume 2

Printed in Italy

Sommario

Presentazione

Questo è il libro della programmazione del 6502 applicata al caso del C64.
Tratta dunque di argomenti generali di programmazione del 6502 (*) in linguaggio macchina:

- *perché programmare in linguaggio macchina;*
- *linguaggio macchina puro e assembler simbolico;*
- *il set di istruzioni del 6502 e i suoi modi di indirizzamento sono studiati in dettaglio;*

e di argomenti specifici del C64:

- *aiuti alla programmazione (monitor, assembler, editor, loader) disponibili per il C64 e loro modi di impiego;*
- *l'ambiente della programmazione assembler e la sua interazione col Basic: come chiamare una subroutine in linguaggio macchina dal Basic, come installarla in memoria, come caricarla;*
- *sono date alcune routine e modi di indirizzamento;*
- *tutti gli esempi sono stati provati sul C64.*

Riassumendo, un apprendimento completo del linguaggio macchina del 6502 e delle sue possibilità, immediatamente applicabili al vostro C64.
I principali argomenti del libro sono i seguenti:

- *il* **capitolo 1** *sviluppa le possibilità dei linguaggi macchina, discute le ragioni per cui usarli e dà una visione generale del set di istruzioni;*
- *il* **capitolo 2** *descrive la rappresentazione interna;*

(*) Il microprocessore del C64 è in effetti un 6510 che non è altro che un 6502 più una porta di input-output. Lo chiameremo quindi "6502" nel seguito.

- *il* **capitolo 3** *studia in dettaglio il set di istruzioni del 6502 con i suoi modi di indirizzamento e li applica ad esempi progressivi che culminano con un programma di disegno animato che non potrebbe funzionare in Basic;*
- *il* **capitolo 4** *vi insegna il modo di usare gli aiuti alla programmazione disponibili: l'assembler simbolico, l'editor, il loader, e il monitor;*
- *il* **capitolo 5** *permette di utilizzare un programma assembler in ambiente Basic sul sistema C64.*

Troverete in appendice, oltre alle soluzioni degli esercizi, una tabella completa del set di istruzioni e una mappa di memoria del C64.

Capitolo 1

Introduzione

LINGUAGGIO ASSEMBLER E LINGUAGGIO MACCHINA

I programmatori usano di solito dei linguaggi evoluti (detti anche: linguaggi ad alto livello). Ciò è giustificato, è infinitamente più facile programmare in linguaggio evoluto che in linguaggio macchina.
Infatti, le istruzioni di un linguaggio evoluto sono sintetiche e potenti mentre il linguaggio macchina non possiede che istruzioni molto elementari. Occorrono dunque, per fare un certo programma, molte più istruzioni in linguaggio macchina che in linguaggio evoluto.
Di fatto, le istruzioni di un linguaggio evoluto sono simboliche, dunque più leggibili del codice binario del linguaggio macchina. Tuttavia vedremo che esistono delle espressioni simboliche del linguaggio macchina.
Inoltre, le istruzioni del linguaggio evoluto sono più vicine alla simbologia matematica che rappresenta il problema da trattare. Ne risulta anche che un programma scritto in linguaggio evoluto ha delle possibilità di essere trasportato da una macchina all'altra senza modifiche, mentre un programma in linguaggio macchina è "incollato" alla macchina considerata e deve essere interamente rifatto se cambiamo macchina.

Perché programmare in linguaggio macchina?

Se ci sono tanti inconvenienti in rapporto ai linguaggi evoluti, perché siamo portati, in alcuni casi, ad utilizzare il linguaggio macchina? Perché, visto che in tutti i casi lo sforzo di programmazione sarà maggiore che in linguaggio evoluto?

Una prima ragione è che, in ogni modo, la macchina non comprende nient'altro che il proprio linguaggio. In più, per poter programmare in linguaggio evoluto, bisogna disporre di un mezzo per tradurre in linguaggio macchina il testo scritto in linguaggio evoluto.
Fortunatamente, questo compito di traduzione può essere affidato ad un programma. È evidente che almeno tale programma deve essere scritto in linguaggio macchina!
Ma anche altri programmi devono essere scritti in linguaggio macchina: sono i programmi principali del sistema operativo, quelli che gestiscono le periferiche.
Una volta che il computer è munito di tali programmi fondamentali e di un traduttore del linguaggio evoluto, si potrebbe pensare che non vi sia più nulla da programmare in linguaggio macchina. E invece sì. Ma, prima, bisogna esaminare brevemente qualche proprietà dei traduttori che spiegherà questa necessità.

Interprete e compilatore

Esistono due tipi di traduttori: gli interpreti e i compilatori.
Un *interprete* traduce ed esegue simultaneamente ciascuna delle istruzioni del programma utente prese successivamente.
Un *compilatore* traduce tutto il programma utente, poi si esegue in blocco il programma tradotto.
Un compilatore è più efficiente di un interprete dal punto di vista dello spazio occupato in memoria e della velocità di esecuzione.
Infatti, l'interprete deve essere presente in memoria contemporaneamente al programma utente che deve interpretare e sovente è più esteso di quest'ultimo, mentre il compilatore è il solo in memoria durante la compilazione: il programma sorgente e il programma tradotto (oggetto) possono essere su disco. In seguito, al momento dell'esecuzione, il programma tradotto è solo in memoria.
Dal punto di vista del tempo, l'interprete è penalizzato dalla necessità di tradurre ogni istruzione prima di eseguirla. In particolare, nel caso di un ciclo – ed è estremamente frequente – le istruzioni del ciclo sono ritradotte ad ogni iterazione mentre sono tradotte una volta per tutte quando si tratta di un compilatore.
Si può dire che il fattore di tempo guadagnato passando (*) – per lo stesso linguaggio evoluto – da un interprete ad un compilatore va da 5 a 50.

(*) Si parla qualche volta di linguaggio compilato o interpretato. Ciò è errato: il fatto di essere interpretato o compilato non è legato al linguaggio (e non dovrebbe entrare nel confronto fra due linguaggi). Così, il Basic è per lo più interpretato ma esistono dei compilatori per il Basic; il Fortran è di solito compilato ma esistono anche delle versioni interpretate.

E nondimeno, anche se si dispone di un compilatore, si può essere portati a dover utilizzare il linguaggio macchina. Infatti, anche un programmatore medio potrà aumentare, utilizzando il linguaggio macchina, la velocità di esecuzione di un fattore da 2 a 20 (tutte queste cifre sono, naturalmente, approssimative). Inoltre, ci sarà un risparmio di spazio in memoria tra il 20 e il 60%.

Facciamo qualche esempio:

Esempio 1. Supponiamo di scrivere in Basic l'istruzione $Y=X^2$. Rari saranno gli interpreti o i compilatori che rimpiazzeranno l'elevamento al quadrato con il calcolo di X*X. La maggior parte dei sistemi calcolerà exp(2*Log(x)), esattamente come se avessimo avuto $Y=X^{2,5}$ che necessita della formulazione exp(2,5*Log(x)).

Esempio 2. Allo stesso modo, se chiediamo una moltiplicazione per 2, il compilatore o l'interprete utilizza una routine di moltiplicazione generale, mentre un programmatore sa che, nel caso particolare del fattore 2, basta effettuare uno scorrimento a sinistra, cioè una sola operazione (0101 in binario 5 in decimale dopo uno scorrimento a sinistra diventa 1010 in binario che vale 10 in decimale). È proprio 5*2.

Esempio 3. Il linguaggio macchina permette, al programmatore, di decidere con precisione l'uso delle risorse interne della macchina, mentre in un linguaggio evoluto questo compito è lasciato all'arbitrio del traduttore. Supponiamo di voler scambiare le variabili T ed U.
In Basic bisogna utilizzare una variabile intermedia:

```
V=U
U=T
T=V
```

ciò che, nell'assembler del 6502, si scriverebbe:

```
LDA U: trasferisce U nel registro interno A (accumulatore)
STA V: memorizza A nel registro V
LDA T
STA U: U contiene ora il valore di T
LDA V
STA T: T contiene ora il valore di U, lo scambio è terminato.
```

Tutti i compilatori farebbero queste operazioni.
Ma il 6502 possiede un altro registro, X che un compilatore non utilizzerebbe perché è normalmente usato per delle indicizzazioni; un pro-

grammatore, sapendo che è libero, per il momento, può utilizzarlo e scrivere:

LDA U: A ← U
LDX T: X ← V
STX U: scambio incrociato
STA T: scambio effettuato

Due istruzioni sono state risparmiate e l'uso del registro intermedio è stato evitato; tali astuzie sono impossibili per un compilatore.
Questi esempi potrebbero moltiplicarsi all'infinito. Si vede che il passaggio dal linguaggio evoluto al linguaggio macchina può fare aumentare la velocità di un fattore da 10 a 1000 (con un interprete).
Il linguaggio macchina sarà dunque utilizzato laddove le prestazioni ottenute in linguaggio evoluto sarebbero insufficienti mentre l'aumento ottenuto col linguaggio macchina le rende compatibili con l'applicazione.

Sul vostro C64,

```
1  10 FOR I = 1024 TO 2023
   20 POKE I,1 : NEXT
   30 FOR I = 1024 TO 2023
   40 POKE I,32 : NEXT

2  10 FOR I = 1024 TO 2023
   20 POKE I,1 : NEXT
   30 PRINT"Clr"
```

Nei due casi, si riempie lo schermo con delle A poi lo si vuota. Nel primo caso, lo svuotamento è fatto alla velocità del Basic. Nel secondo caso, l'istruzione PRINT "Clr" chiama una routine in linguaggio macchina che effettua lo svuotamento: potete notare la differenza di velocità!
Il linguaggio macchina deve inoltre essere utilizzato per accedere ad alcune risorse della macchina inaccessibili in linguaggio evoluto, per esempio per disabilitare alcune routine di interruzione. È necessario per sostituire le proprie routine di sistema col fine di modificare un particolare comportamento della macchina.

Linguaggio macchina e assemler simbolico

Quando si parla di linguaggio macchina, si indicano in effetti due cose ben differenti, o talora due "livelli" di linguaggio.
Il linguaggio macchina propriamente detto è in codice binario: per esempio, sul 6502, caricare in accumulatore (il principale registro di

calcolo del 6502) il contenuto dell'indirizzo di memoria 1000 si scrive: 1010 1101 1110 1000 0000 0011.
È difficile da maneggiare: cercate di dettare l'istruzione qui sopra al telefono! È per questo che l'impiego dell'esadecimale o dell'ottale apporta una grande semplificazione; in esadecimale che è universalmente impiegato coi microprocessori ad 8 bit, si raggruppano i bit 4 a 4 e ogni quartetto è rimpiazzato da una cifra del sistema di numerazione a base 16, da 0 a 9 e da A ad F per le cifre maggiori di 9. Così l'istruzione qui sopra si scriverebbe AD E8 03.
Tuttavia, questa semplificazione non cambia l'essenziale e gli inconvenienti di programmare in questa forma sono grandi: il programmatore deve gestire tutti gli indirizzi delle proprie variabili e conoscere a memoria (o consultare una tabella) il codice binario o esadecimale di tutte le istruzioni.
È per questo che si è introdotto un altro livello di linguaggio, chiamato assembler simbolico.
Tutte le istruzioni in linguaggio macchina comportano due parti:

1. generalmente su un byte, un numero binario che indica il tipo dell'operazione da effettuare (è il codice operativo);
2. se occorre, uno o più byte che formano l'indirizzo di una cella di memoria sul contenuto della quale si effettuerà l'operazione.

Così, nell'istruzione qui sopra (AD E8 03) che carica in accumulatore il contenuto della cella di memoria 1000 su un 6502, AD è il codice operativo; significa "carica in accumulatore", mentre i due byte E8 03 formano una codifica (che spiegheremo più avanti) dell'indirizzo 1000. Queste due parti si ritrovano in un'istruzione espressa in assembler simbolico.

1. Il tipo di operazione è rappresentato da un piccolo nome detto "mnemonico" perché ricorda la funzione dell'operazione. Per esempio, un'addizione sarà designata nella maggior parte degli assembler con "ADD". Il guaio è che i codici mnemonici sono delle abbreviazioni dei termini inglesi che descrivono l'operazione: dunque sono pienamente mnemonici solo per quelle persone che conoscono l'inglese.
   Per la nostra istruzione di caricamento in accumlatore, vista qui sopra, lo mnemonico in 6502 sarà LDA, abbreviazione di Load Accumulator, perché inviare un valore in un registro si dice generalmente "to load", caricare. Con un po' di pratica, ci si abitua molto velocemente agli mnemonici inglesi.
2. Quando l'istruzione agisce su una certa cella di memoria (si dice che comporta un operando), questa cella di memoria è rappresentata sia dal suo indirizzo, ma scritto in decimale o alcune volte da un nome simbolico che ha esattamente lo stesso ruolo di un identifica-

tore di variabile nel linguaggio evoluto. Così la nostra solita istruzione si scriverà:

LDA 1000 oppure LDA TASSO

se la grandezza che maneggiamo è un tasso di interesse in vista di calcoli finanziari: come nel linguaggio evoluto, è consigliabile prendere dei nomi che ricordino il ruolo svolto dalle variabili maneggiate.

Si vede dunque quanto la forma simbolica sia più semplice da utilizzare della forma binaria (anche se scritta in esadecimale).
La successione è la seguente: linguaggio macchina, assembler simbolico, linguaggio evoluto e l'impiego dell'assembler simbolico si inserisce a metà strada fra il linguaggio macchina e il linguaggio evoluto.
Come in un linguaggio evoluto, disponiamo di simboli e di variabili, ma, come nel linguaggio macchina, bisogna scomporre al massimo le operazioni.
Ne risulta che, in tutti i casi, un programma sarà preparato sotto forma simbolica perché è più facile da maneggiare e da rileggere. A seconda dell'hardware e del software di cui si dispone questa sarà la sola da preparare oppure no.
Però, in ogni caso, il computer non potrà che eseguire la forma binaria. Bisogna dunque, come per il linguaggio evoluto, effettuare una traduzione dal programma in assembler simbolico al codice binario o esadecimale.
Questa traduzione si fa sia a mano, sia con l'aiuto di un programma che si chiama "assembler". La fase di traduzione si chiama spesso assemblaggio, da cui il nome di assembler attribuito, un po' impropriamente, al linguaggio simbolico: l'assembler propriamente detto è il traduttore.
Per poter funzionare un assembler simbolico abbisogna di differenti elementi sia hardware che software:

- occorrono una tastiera ed uno schermo alfanumerici;
- occorre uno spazio di memoria sufficiente (eventualmente con disco o cassetta);
- occorre avere il programma assembler stesso, coi suoi programmi ausiliari come un editor di testi, un loader, un monitor per la messa a punto.

Se il computer non dispone di questi elementi (è il caso, per esempio, dei microcomputer su di una sola scheda come i KIM-1 o il MEK 6800D2, dove non si ha che una tastiera esadecimale e un display a sette segmenti, e dove la memoria disponibile non permette di usare che un monitor assai ridotto) ci sono due soluzioni:

- fare l'assemblaggio su di un altro computer più grande, poi inserire il risultato dell'assemblaggio con la tastiera esadecimale. Si parla allora di un assembler incrociato (cross assembler);
- fare l'assemblaggio a mano. È un compito noioso e con grande rischio di errori. È accettabile solo per piccolissimi programmi (meno di 100 o 200 istruzioni) come i programmi didattici.

Questo compito non presenta tuttavia alcuna difficoltà di principio: si sostituisce il codice mnemonico dell'istruzione col codice esadecimale corrispondente e l'operando, se esiste, col suo indirizzo esadecimale. Disponiamo, per questo, di una tabella dei codici operativi in esadecimale fornita dal costruttore. Solamente i calcoli di indirizzo sono un po' delicati: devono essere fatti con cura perché errori in questo punto sono fatali. Altrimenti la traduzione viene effettuata istruzione per istruzione poiché esiste corrispondenza biunivoca tra le istruzioni del linguaggio simbolico e la loro codifica in binario.
L'uso di un assembler è infinitamente più comodo. Abbiamo visto i due campi principali di un'istruzione simbolica: il codice mnemonico e l'operando. L'assembler ammette altri due campi: il campo label (il più a sinistra) che permette di dare un nome ad un'istruzione al fine di ritrovarla nel programma se deve essere l'arrivo di un'istruzione di salto, e il campo commento che permette di documentare il programma. Così la nostra istruzione si potrebbe scrivere:

INIZIO LDA TASSO; ricerca del tasso di interesse

Le regole di sintassi differiscono leggermente da un assembler all'altro; spesso i diversi campi sono separati da uno spazio. Se una linea non contiene label non deve iniziare dalla prima colonna. Il campo commento è sovente preceduto dal punto-e-virgola. Se una linea inizia con un ";" significa che è interamente di commento.

; FASE DI INIZIALIZZAZIONE

L'assembler simbolico procede esattamente come una persona che esegue l'assemblaggio a mano:

- nella prima fase, separa e riconosce i quattro campi dell'istruzione;
- in seguito il codice mnemonico è tradotto (esame della tabella dei codici operativi). Un messaggio di errore viene emesso se il codice non è riconosciuto;
- il campo operando è il più delicato: bisogna tradurre il o i simboli utilizzati in un indirizzo. Ma vi sono diversi modi per specificare un indirizzo: ci sono più modi di indirizzamento. Ne riparleremo; diciamo solamente qui che l'indirizzo può essere semplice (è semplicemente l'indirizzo desiderato) o composto; per esempio, l'indirizzo effettivo sarà ottenuto aggiungendo il contenuto di un registro indice (indirizzamento indicizzato).

C'è dunque, in questa fase, un'analisi sintattica per riconoscere il modo di indirizzamento scelto dal programmatore e un calcolo dei byte di indirizzo da implementare.
Questo calcolo esige la sostituzione dei simboli che rappresentano gli operandi con il valore che costituisce l'indirizzo in questione. Per questo, l'assembler si vale di una tabella, la symbol table, che stabilisce questa corrispondenza. Può presentarsi una difficoltà: durante l'assemblaggio di un'istruzione, può darsi che il valore di un simbolo che interviene nel calcolo dell'indirizzo da implementare sia ancora sconosciuto; si dice che il riferimento è "in avanti"; l'esempio più banale è quello di un'istruzione di salto più avanti nel programma:

```
QUI JMP LÀ; SALTA ALL'ISTRUZIONE LÀ
     –
     –
     –
     –
LÀ   –
     –
     –
```

Durante l'assemblaggio dell'istruzione **QUI**, l'indirizzo di **LÀ** è ancora sconosciuto. La soluzione dei riferimenti in avanti è un punto delicato dell'assemblaggio, ma se non se ne disponesse, l'assembler simbolico perderebbe molta della sua utilità.

DIRETTIVE

Durante l'assemblaggio, l'assembler aggiorna un contatore chiamato contatore ordinale simboleggiato da '*' e che contiene l'indirizzo dell'istruzione in corso di assemblaggio.
Questo contatore è inizializzato da un'istruzione del tipo *=valore oppure ORG valore. Un'istruzione del genere si chiama *direttiva*. È un'istruzione non eseguibile che ha lo stesso ruolo di una dichiarazione di un linguaggio evoluto.
L'assembler ammette altre direttive, come END (fine programma) oppure FILE (assemblare partendo da un file).
Alcune direttive hanno un ruolo nell'impaginazione del listato, come PAGE (salta alla pagina nuova), oppure SKIP n (salta n linee).
Le direttive principali inizializzano dei simboli o riservano spazio di memoria:

- TOT RMB 5 riserva 5 byte in memoria a partire da TOT. Può anche scriversi: TOT *=*+5;
- TOT BYTE $FF mette, nella cella di memoria TOT, il valore esadecimale (segnalato da $) FF, cioè 1111 1111.

Quest'ultima direttiva non deve essere confusa con:

- TOT = $1000 che assegna al simbolo TOT il valore 1000, cioè TOT rappresenta l'indirizzo 1000 in esadecimale.

Queste due direttive possono combinarsi:

```
* = $1000 ; TOT rappresenta l'indirizzo 1000
TOT BYTE $FF ; a questo indirizzo c'è il valore FF esa.
```

Gli assembler sofisticati permettono delle espressioni aritmetiche nel campo operando:

```
TOT = 1000
LDA TOT+5 ; carica il contenuto dell'indirizzo 1005.
```

LISTATI

La parte di destra di ogni linea di listato, che corrisponde ad un'istruzione, riproduce questa istruzione come l'abbiamo scritta. A sinistra si trovano in successione:

- un numero di linea;
- l'indirizzo dell'istruzione (in esadecimale);
- il codice esadecimale su uno o più byte corrispondente all'istruzione:

Esempio

```
0001 0000            *     = $2000  ; origine in 2000
0002 2000            TASSO = $1000  ; tasso in 1000
0003 2000 AD 00 10   DEP     LDA
0004 2003 A2 05              LDX #5
```

Le prime due linee non hanno codice assemblato poiché sono delle dichiarazioni. Il listato termina con dei messaggi diagnostici e con la lista della tabella dei simboli se è richiesta.

Esempio

Tabella dei simboli

SIMBOLO	VALORE	RIFERIMENTO
TASSO	1000	2000

Macroassembler e macroistruzioni

Si dice macroassembler un assembler che permette di scrivere:

```
SCAMBIO  MACRO  T,U
         LDX    T
         LDA    U
         STA    T
         STX    U
         ENDM
```

Così si chiama la definizione o il prototipo della macro SCAMBIO. Diventa allora possibile scrivere, in più punti del programma:

```
QUI     SCAMBIO     TASSO1      TASSO2
```

ciò avrà lo stesso effetto di:

```
QUI   LDX   TASSO1
      LDA   TASSO2
      STA   TASSO1
      STX   TASSO2
```

In pratica è un modo di arricchire il set di istruzioni.
Ci sono, segnatamente dal punto di vista della gestione dei parametri formali, delle grandi analogie con i sottoprogrammi così come li conosciamo per i linguaggi evoluti. C'è però una differenza importante: nel richiamare un sottoprogramma, ad ogni chiamata, c'è un'istruzione di salto, quindi lo spazio di memoria occupato dal sottoprogramma è economizzato, mentre ogni volta che si fa riferimento ad una macro, una copia della macro è implementata e, ogni volta, occupa lo spazio di memoria corrispondente.
Ciò è dovuto al fatto che le macro sono utilizzate durante la fase di assemblaggio, e non durante l'esecuzione.
Il loro interesse è tuttavia grandissimo, tanto che le definizioni di macro possono essere annidate e possono essere gestite delle biblioteche di macro. Ma, sfortunatamente, i macroassembler sono abbastanza rari su microelaboratori a buon mercato.

Assemblaggio condizionato

Un'altra facilitazione offerta da alcuni assembler permette di scrivere:

istruzioni 1
IF LUNG > 10
istruzioni 2
ENDIF
istruzioni 3

Se al momento dell'assemblaggio, la condizione (qui, LUNG > 10) è soddisfatta, le istruzioni 2 saranno assemblate. Altrimenti, saranno assemblate solo le istruzioni 1 e le istruzioni 3. Ciò permette di preparare un programma molto generale e di assemblarne versioni differenti a seconda delle necessità.

POSSIBILITÀ DEL LINGUAGGIO MACCHINA

In questa seconda parte, stiamo ora per esaminare le nozioni comuni a tutti i microprocessori a 8 bit del mercato, insistendo tuttavia più sul 6502 che equipaggia il C64 (il C64, in realtà, possiede un 6510 che è una versione del 6502 che ha esattamente lo stesso set di istruzioni).
Vedremo i principali registri, le differenti istruzioni utilizzate su tutti i microprocessori, così come i modi di indirizzamento.
Faremo, nello stesso tempo, un breve confronto dei differenti microprocessori.

I registri

La programmazione in linguaggio macchina, ed è lì il suo interesse, dà al programmatore accesso alle risorse interne della macchina. Queste risorse sono essenzialmente la memoria che bisogna gestire indirizzo per indirizzo, e i registri interni del microprocessore.
I registri non sono nient'altro che memoria, ma a cortissimo termine, destinati a memorizzare dei dati quando dei calcoli devono essere effettuati su questi ultimi.
Questa situazione è del tutto analoga a quella di un aiuto-contabile che effettua dei calcoli: egli possiede un quaderno sul quale si trova la lista delle operazioni che deve svolgere; i dati sono stati presi e i risultati saranno riportati su un quaderno o un libro di contabilità.
Ma, per effettuare ogni operazione, l'aiuto-contabile si serve di un bloc-notes di cui straccia i fogli non appena sono pieni.
Ebbene, ciò che nel calcolatore svolge il ruolo dei differenti quaderni è la memoria, e sappiamo bene che contiene sia la lista delle operazioni da effettuare (il programma) che i dati da manipolare.
E ciò che, nel calcolatore, svolge il ruolo di bloc-notes, conservando le informazioni intermedie sulle quali si fanno i calcoli, è l'insieme dei registri interni del microprocessore.
Se si vogliono comprendere le istruzioni del microprocessore, bisogna conoscere la struttura dei suoi registri interni poiché tutte le istruzioni del microprocessore ne fanno intervenire almeno uno.

Presenteremo, qui, i registri interni del 6502 e citeremo qualche differenza con altri microprocessori.
La **figura 1** rappresenta i registri programmabili (cioè manipolabili da programma) del 6502.

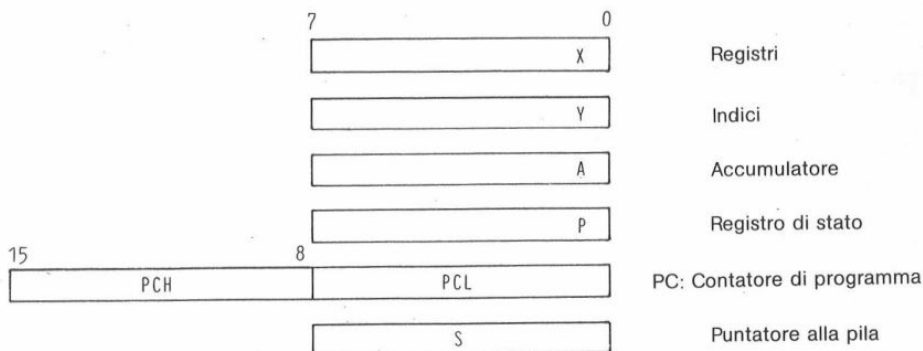



Fig. 1. Registri programmabili del 6502/6510.

I due registri più importanti sono **A** e **PC**.
A è **l'accumulatore**, a 8 bit. È il registro sul quale sono svolte la maggior parte delle istruzioni aritmetiche del 6502 che sono nella forma: A ← A operazione M, cioè si fa l'operazione fra A e la cella di memoria M; il risultato è di nuovo il contenuto dell'accumulatore.
PC è il **contatore di programma** (program counter). Torniamo alla nostra analogia con l'aiuto-contabile. Se ha molte operazioni da fare, è probabile che annoterà, su un bloc-notes, a quale operazione è giunto.
Ebbene, è **PC** che svolge questo ruolo per il microprocessore: in ogni momento, **PC** contiene l'indirizzo di memoria della prossima istruzione da eseguire. **PC** è considerato come registro programmabile; infatti, vi sono delle istruzioni che lo modificano: le istruzioni di salto e di biforcazione. **PC** è un registro a 16 bit (l'unico del 6502).
I registri **X** e **Y** sono dei registri **indice**, cioè il loro contenuto è suscettibile di aggiungersi ad un indirizzo. Ne riparleremo a proposito dell'indirizzamento indicizzato.
Il **registro P** (Processor status register) riunisce i 7 bit (il bit 5 è inutilizzato) il cui valore rappresenta una condizione che si è verificata all'interno del microprocessore nello svolgimento di un'operazione precedente. Questi bit si chiamano flag (indicatori di stato) da cui il nome del registro che li raggruppa: **registro di stato della macchina**.

La **figura 2** rappresenta i diversi bit del registro di stato del 6502, del 6800 e dello Z80. La maggior parte dei flag si trova su tutti i microprocessori.

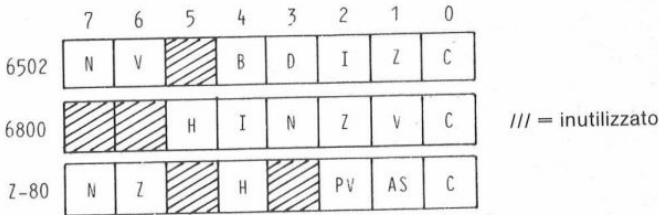



Fig. 2. Flag di stato.

Dettagliamo ora il ruolo dei differenti bit.
Infatti, si deve distinguere tra flag che memorizzano un evento successo durante un'operazione decimale (esempio: riporto) e flag che influenzano il comportamento futuro del microprocessore (esempio: modo decimale).

N è il bit di segno: vale 1 se l'ultimo risultato calcolato è negativo.

V è il flag di superamento di capacità (OVERFLOW): vale 1 quando si è prodotto un superamento di capacità.

Z è il flag di zero: vale 1 se l'ultimo risultato ottenuto è nullo (vale 0 se il risultato è non nullo).

C è il flag di riporto (carry): vale 1 se l'ultima operazione ha prodotto un riporto.

B è il flag di break: vale 1 quando il 6502 ha appena effettuato una istruzione BRK (interruzione software).

I è un flag di modo: quando vale 1 le interruzioni sono disabilitate.

D è il flag di modo decimale. Quando vale 0 il 6502 opera in binario: per esempio, 19 + 1 = 1A; quando vale 1, il 6502 opera in modo decimale codificato binario: per esempio 19 + 1 = 20.

Flag particolari del 6800 e dello Z80

Il 6800 e lo Z80 non possiedono modo decimale. La somma 19 + 1 dà sempre 1A. Se si ritiene di essere in modo decimale, occorre un'istru-

zione speciale di aggiustamento del risultato (1A deve diventare 20). Questa istruzione utilizza il flag **H** (riporto intermedio) che vale 1 allorché ci sia stato un riporto del bit 3 verso il bit 4, e il flag **AS** che vale 0 se l'operazione effettuata era una addizione e 1 se era una sottrazione. Questo flag non esiste sul 6800 che può eseguire l'aggiustamento solo dopo una addizione.
Infine, il flag **PV** dello Z80 è un flag di superamento di capacità (V) ma che svolge talvolta il ruolo di bit di parità.
Il valore di questi flag è assegnato sia automaticamente in funzione di ciò che avviene durante un'operazione, sia tramite alcune istruzioni speciali in grado di forzare l'1 o lo 0. Sono inoltre suscettibili di essere testati da istruzioni di biforcazione.
Benché di ruolo generalmente analogo da un microprocessore all'altro, i flag possono essere trattati differentemente per certi dettagli. Il programmatore deve esaminare attentamente l'effetto di ogni istruzione sui flag (che è descritto nella documentazione) perché anche un dettaglio infinitesimo cambia il comportamento di un programma. È questa una delle schiavitù della programmazione in linguaggio macchina.
Si può dire che due microprocessori col medesimo set di istruzioni, ma che agiscano in maniera diversa su un flag, siano in effetti due macchine diverse e alcuni programmi potrebbero risultare incompatibili passando dall'uno all'altro. Ed è già successo! Delle copie non autorizzate dell'8080 avevano delle differenze infinitesime sui flag; alcuni programmi che giravano sull'8080 originale si "piantavano" sulla copia!
Il **registro S** è il puntatore alla pila (Stack Pointer). Tutti i microprocessori devono gestire una pila, cioè una zona di memoria che obbedisce alla regola LIFO (last in firs out = ultimo entrato primo uscito). Ciò è necessario in particolare per i sottoprogrammi e le interruzioni. Il 6502 dispone, per questo, di un registro, il puntatore alla pila che punta alla sommità della pila e che è aggiornato ad ogni operazione sulla pila. Il registro è di 8 bit, ciò che limita la pila a una dimensione di 256 byte (è ampiamente sufficiente).
Per ogni operazione sulla pila, il 6502 invia come indirizzo esadecimale 100 + il contenuto di S, quindi la pila è compresa fra 100 e 1FF. Il 6800 e lo Z80 hanno un puntatore alla pila di 16 bit, quindi la pila può essere ovunque e di lunghezza qualsiasi (questo non serve a nulla).
Come altre differenze da segnalare, notiamo che il 6800 ha due **accumulatori A e B** aventi circa le stesse potenzialità e che, in luogo di due registri indice di 8 bit, ne ha uno solo di 16 bit, **IX**.
L'esperienza mostra che il modo di procedere del 6502 è più flessibile. Lo Z80 ha una filosofia un po' differente. Ha un numero enorme di registri interni in più del 6502 e del 6800, di cui molti servono da registri di manovra: è più orientato a "rimestare" nei registri interni che ad utilizzare la memoria.

Notiamo anche che i microprocessori dispongono in effetti di altri registri di cui non abbiamo parlato perché non sono accessibili al programmatore. È il caso del registro I che contiene il codice operativo dell'istruzione in corso: è ovvio che non deve essere modificato da programma!

Le istruzioni

Ma, di quali istruzioni si dispone per manipolare tutti i registri? Si può proporre qualsiasi tipo di classificazione. Per conto nostro, proporremo la seguente.

1. *Istruzioni di trasferimento di informazione (senza trattamento):*
   - *trasferimento tra registri;*
   - *trasferimento tra registri e memoria;*
   - *istruzioni concernenti la pila.*
2. *Istruzioni aritmetiche e logiche:*
   - *operazioni unarie (1 operando);*
   - *operazioni binarie (2 operandi).*
3. *Operazioni sui flag di stato:*
   - *operazioni incondizionate;*
   - *confronti.*
4. *Istruzioni di salto:*
   - *istruzioni di biforcazione;*
   - *salti incondizionati e istruzioni diverse.*
5. *Istruzioni di ingresso-uscita:*
   - *(assenti sul 6502 e sul 6800).*

1. Istruzioni di trasferimento

Trasferimento fra registri

Il 6502 possiede sei istruzioni di questo tipo:

TAX TXA TAY TYA TSX TXS

TAB si legge "trasferisci da A in B".

Trasferimento tra registri e memoria

Il trasferimento da memoria in un registro si chiama caricamento (load); il trasferimento da registro in memoria si chiama memorizzazione (store). Sul 6502, i tre registri influenzati da queste operazioni sono **A, X** e **Y**, da cui:

- **LDA** M trasferisce da M in A: A ← M
- **STA** M M ← A
- **LDX** M (X ← M) **STX** M (M ← X) **LDY** M (Y ← M) **STY** M (M ← Y)

Queste operazioni sono fra le più importanti. Si noterà che, sul 6502, **S** e **P** non dispongono di istruzioni di questo tipo: bisogna procedere in modo indiretto in caso di bisogno.

Trasferimenti riguardanti la pila

Si possono mettere o togliere dalla pila l'accumulatore o il registro di stato. Se chiamiamo **MS** l'indirizzo di memoria al quale punta il puntatore della pila, **MS** è la prima cella libera sotto la cima della pila e **MS+1** è la cima della pila. Allora:

- **PHA** MS ← A; S=S–1
- **PLA** S=S+1; A ← MS
- **PHP** MS ← P; S=S–1
- **PLP** S=S+1; P ← MS

Si noterà che TAP che non esiste sul 6502 (P ← A) si può simulare con la sequenza PHA; PLP e che anche PHP; PLA equivale a A ← P.

2. Istruzioni aritmetiche e logiche

Ecco le istruzioni che permettono di effettuare i calcoli o di elaborare nuove informazioni a partire da dati di partenza.

Operazioni binarie

Sono così chiamate le istruzioni con due operandi. Nel caso del 6502, uno dei due operandi è l'accumulatore A, l'altro è il contenuto di una cella di memoria specificata. Il risultato è rimesso nell'accumulatore (il vecchio valore è perso), esempio:

AND M esegue A ← (A) and (M)

Sul 6800, bisogna precisare quale dei due accumulatori A o B è implicato; esempio:

LDA B TIZIO
AND A CAIO

Le operazioni logiche AND, ORA, EOR lavorano su ciascun bit. Eseguono rispettivamente l'**AND**, l'**OR** e l'**OR esclusivo** dei due operandi, conformemente alle tavole di verità:

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

OR ESCL	0	1
0	0	1
1	1	0

Queste operazioni possono servire a forzare a 1 o a 0 un bit di una cella di memoria se un operando è una costante.

Esempio

Forzare a 1 il bit 5 della memoria M (lasciando gli altri inalterati):

```
LDA  M
ORA  #$20
STA  M
```

Nella seconda istruzione qui sopra, il segno # significa che il valore col quale si fa l'OR dell'accumulatore è proprio la costante 20 e non il valore che si trova all'indirizzo 20. Ciò si chiama indirizzamento immediato. Ne riparleremo.
Il $ significa che la costante è espressa in notazione esadecimale 20 hex = 0010 0000 binario, non c'è che il bit 5 a 1.
Per forzare a 0, bisogna fare un AND con un bit a 0. Così, AND#$BF forza a zero il bit 5. L'EOR inverte il valore dei bit:

```
EOR  #$20  inverte il bit 5
EOR  #$FF  inverte tutti i bit
```

in pratica calcola il complemento a 1.
Arriviamo ora alle operazioni aritmetiche. Sembra che il 6502 ne abbia poche, ma è da vedere.
Le operazioni **ADC** e **SBC** sembrano porre un problema. Infatti esse fanno intervenire il riporto (flag C) che si aveva prima dell'inizio dell'operazione:

così, **ADC** svolge l'operazione $A \leftarrow A+M+C$

Quando non si vuole che il riporto intervenga, occorre, sul 6502, annullare precedentemente quest'ultimo; esiste un'istruzione per questo: **CLC**. Sul 6800 si dispone di **ADC** ma anche di **ADD** cioè addizione senza riporto.
L'utilità di ADC deriva dalla necessità di manipolare abbastanza spesso dei numeri su più byte perché non si può accontentarsi di numeri compresi fra 0 e 255.
Se si devono sommare due numeri in doppia precisione, si sommeranno successivamente i byte cominciando dai meno significativi. Per la prima somma si usa ADD o CLC e ADC. Ma per le successive, si usa ADC perché, ovviamente, bisogna tenere conto del riporto che si è generato precedentemente.
Il principio è il medesimo per la sottrazione. Ma, poiché l'effetto di **SBC** è: $A \leftarrow A-M-\overline{C}$, cioè si sottrae il complemento del valore che il

riporto ha all'inizio dell'istruzione. Se si vuole una sottrazione normale, occorre fare **SEC** (C ← 1), poi **SBC**.
Un'altra particolarità di **ADC** e **SBC** del 6502 è che queste istruzioni funzionano in due modi: a seconda che il flag D sia a 0 o a 1, i numeri manipolati sono considerati come binari (19+1=1A) o decimali codificati binario (ogni byte va da 00 a 99 e 19+1=20).
Il 6502 è il solo che possegga tale possibilità e ciò gli conferisce una grande efficacia per un'applicazione del tipo terminale di punto di vendita, per esempio. Gli altri microprocessori calcolano sempre in binario, ma utilizzano una istruzione **DAA** dopo ogni operazione allorché, sul 6502, basta mettersi in modo decimale una volta per tutte. In più, sul 6800, l'aggiustamento decimale è operativo solo dopo un'addizione: quindi bisogna effettuare la sottrazione in modo indiretto.

Operazioni unarie

Queste operazioni hanno un solo operando che può essere l'accumulatore, i registri indice X e Y, o una cella di memoria M. Sono gli scorrimenti e le rotazioni (su **A** o **M**), vedi **figura 3**.

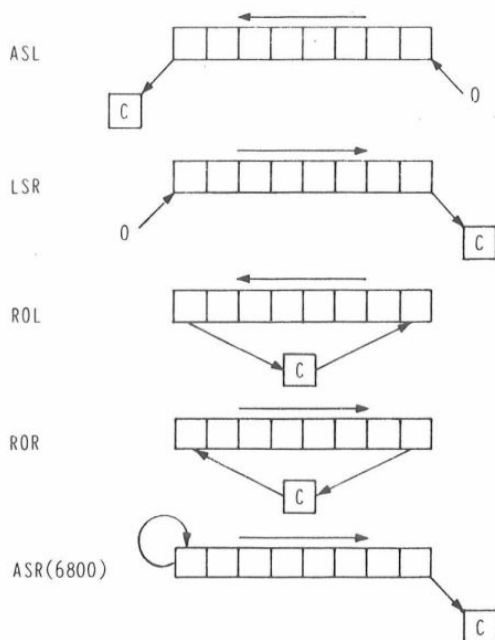



Fig. 3. Scorrimenti e rotazioni.

Esempio

ROL A rotazione a sinistra dell'accumulatore
ASL TIZIO scorrimento a sinistra della cella TIZIO

Per gli scorrimenti, il bit che esce dall'operando va nel riporto ed è uno 0 che rientra dall'altra parte. Le rotazioni si eseguono su 9 bit: l'operando + il riporto.
Incrementi e decrementi

INC M	(M ← M+1)	**DEC M**	(M ← M–1)	agisce su una memoria
INX	(X ← X+1)	**DEX**	(X ← X–1)	agisce sul registro X
INY	(Y ← Y+1)	**DEY**	(Y ← Y–1)	agisce sul registro Y

Non esiste **INC A** nel 6502 (lo si può fare con **CLC** e **ADC** # 1) mentre esiste sul 6800. Il 6800 ha in più del 6502: **COM** (complemento a 1), **NEG** (complemento a 2) e **CLR** (azzera).

3. Operazioni sui flag

Citeremo dapprincipio le operazioni di forzamento a 1 o a 0 di un flag:

- **CLC** riporto = 0
- **SEC** riporto = 1
- **CLV** superamento capacità = 0 (non esiste sul 6502)
- **CLD** modo decimale = 0 (D=0 → modo binario)
- **SED** modo decimale = 1 (D=1 → modo decimale)
- **CLI** I = 0 interruzioni abilitate
- **SEI** I = 1 interruzioni disabilitate

Confronti

Queste istruzioni effettuano un'operazione, ma il risultato non è memorizzato in un registro; si utilizza semplicemente il modo in cui l'operazione posizionerà i flag per degli ulteriori test. L'operazione è detta virtuale.

CMP confronto fra l'accumulatore e una cella di memoria
CPX confronto fra X e una cella di memoria
CPY confronto fra Y e una cella di memoria

Il fatto che l'operazione sia virtuale permette dei confronti in serie:

Esempio

LDA	carattere da tastiera
CMP #'*'	è un asterisco?
BEQ AST	salta ad AST se sì
CMP #'+'	è un più?
BEQ PIÙ	

I caratteri sono rappresentati simbolicamente da 'carattere'. Un'altra istruzione virtuale del 6502 permette di testare dei bit isolati. È **BIT** che effettua l'**AND** virtuale fra l'accumulatore e una cella di memoria.

Esempio

Se si vuole testare il bit 5, una maschera è caricata con un unico 1 al bit 5:

LDA #%00100000
BIT TIZIO

Se il bit 5 di **TIZIO** è 1, l'**AND** avrà risultato non nullo e il flag **Z** sarà 0.
In più, **BIT** copia i bit 7 e 6 della memoria sui flag **N** e **V**. Quindi i bit 6 e 7 possono essere testati senza caricare la maschera nell'accumulatore.

4. Istruzioni di salto

Biforcazioni

Queste istruzioni sono nella forma:
test di una condizione (su un flag). Se questa condizione è verificata, allora salta all'indirizzo indicato (si dice che si ha una biforcazione), altrimenti prosegui in sequenza.

Esempio

QUI BCC LÀ; salta a LÀ se C=0
LÀ ...

- **BCC**: salta se il riporto è 0 (C clear)
- **BCS**: salta se il riporto è 1 (C set)
- **BVC**: salta se l'overflow è 0 (V clear)
- **BVS**: salta se l'overflow è 1 (V set)
- **BMI**: salta se meno (N=1)
- **BPL**: salta se più (N=0)
- **BEQ**: salta se uguale (Z=1)
- **BNE**: salta se non uguale (Z=0)

Altri processori possono testare altre condizioni. Il 6800 testa alcune combinazioni di più flag, ciò che facilita il confronto di numeri con segno o senza segno.
Queste situazioni sono fondamentali per tutti i test e naturalmente per eseguire dei cicli. Il linguaggio assembler non ha istruzioni globali per i cicli, bisogna scriverle esplicitamente.

Esempio

Ripetere 10 volte qualche cosa:

```
       LDX #0                               LDX #10
CICLO   .                            CICLO   .
        .  } istruzioni da ripetere          .  }
        .                                    .
       INX                                  DEX
       CPX #10                              BNE CICLO
       BNE CICLO
       (a)                                  (b)
```

Si vede che la soluzione (b) risparmia una istruzione. Si ha spesso interesse a fare diminuire il registro che serve come contatore, ma (a) è più chiara al momento della rilettura del programma.
Lo Z80 possiede delle istruzioni che riuniscono il decremento, il test e la biforcazione come DJNZ (Decrement and Jump if Not Zero).

Altre istruzioni di salto e diverse

JMP è l'istruzione di salto incondizionato: si salta in ogni caso senza testare alcuna condizione.
JSR è l'istruzione di chiamata di un sottoprogramma. Il 6502 non ha che questa: la chiamata è incondizionata mentre lo Z80 ha delle istruzioni di chiamata condizionata in cui la chiamata viene fatta solo se una condizione è verificata.
Tutti i microprocessori hanno almeno due istruzioni di ritorno: **RTS** ritorno da sottoprogramma e **RTI** ritorno da interruzione.
Quando si chiama un sottoprogramma, l'indirizzo di ritorno è memorizzato nella pila. **RTS** ha il solo effetto di togliere dalla pila due byte e inviarli al **PC**. Sul 6502, **RTI** toglie dalla pila il registro **P** che era stato messo in pila con **PC al momento dell'interruzione.**
Le altre istruzioni di questa categoria dipendono dal microprocessore. La maggior parte dei microprocessori ha una istruzione NOP e una **BRK. NOP** (No Operation) non fa nessuna operazione. Serve per delle temporizzazioni o per correggere dei programmi quando non si vuole ricompilare il tutto. **BRK** simula, da software, una interruzione. Sul 6800 si chiama **SWI**.

4. Istruzioni di ingresso-uscita

Né il 6502 né il 6800 hanno istruzioni di questa categoria. Essi utilizzano, infatti, la tecnica di ingresso-uscita progettata in memoria nella quale i registri d'interfaccia alle periferiche sono visti dal microprocessore, esattamente come se fossero delle celle di memoria.

Tutti gli ingressi-uscite possono allora essere effettuati con l'aiuto delle istruzioni abituali di manipolazione della memoria: **LDA** per una lettura e **STA** per una scrittura.
Questa concezione, che è in uso fin dalla prima apparizione dei microprocessori, ha numerosi vantaggi: di questi il principale è di rendere banali le operazioni di ingresso-uscita.
Queste appaiono ora come delle operazioni ordinarie mentre, nel caso di elaboratori classici o mini, c'era tutto un set di istruzioni dedicate; queste istruzioni erano le più delicate e costituivano la bestia nera dei programmatori.
Il vantaggio è doppio: il set di istruzioni è semplificato e, d'altra parte, le combinazioni dei codici operativi lasciate libere fra le 256 possibilità possono essere messe a profitto per offrire dei modi di indirizzamento e delle istruzioni di memoria più potenti. Questa potenza beneficia anche gli ingressi-uscite che sono banalizzati.
Lo Z80 è rimasto alla concezione dell'ingrsso-uscita svolto tramite un set di istruzioni speciali che comprende **IN** (ingresso da una periferica in un registro) e **OUT** (uscita da un registro verso una periferica) e le loro varianti. Lo Z80 ha inoltre un insieme di istruzioni di trasferimento di blocchi, vale a dire delle istruzioni ripetitive che effettuano automaticamente il trasferimento propriamente detto, il decremento del registro di conteggio (B) e il test di biforcazione.
Un altro vantaggio degli ingressi-uscite progettati in memoria è di riportare la complessità di certe operazioni di ingresso-uscita nella logica di un'interfaccia specializzata, da cui l'interesse di disporre di una famiglia di interfacce associate a un dato microprocessore. La ricchezza di questa famiglia costituisce spesso un miglior criterio di scelta che l'esame del solo microprocessore.

Modi di indirizzamento

Il modo di indirizzamento è la maniera in cui l'indirizzo della cella è specificato in una istruzione che fa riferimento alla memoria.
La flessibilità dei modi di indirizzamento disponibili conta per la potenza di un microprocessore più del set di istruzioni stesso.
Un primo modo che può presentarsi consiste nel fornire, dopo il codice operativo, il dato medesimo sul quale si vuole operare. Non è, propriamente, un modo di indirizzamento, poiché non si fornisce l'indirizzo ma il dato. Lo si chiama tuttavia tradizionalmente indirizzamento immediato e, in assembler simbolico, è segnalato da un #:

LDX #$FF

($ vuol dire esadecimale) che viene assemblato in

A2	**FF**
codice operativo	dato

Naturalmente si suppone che il dato sia conosciuto al momento dell'assemblaggio; anche se è fornito sotto forma simbolica, deve essere una costante.

Esempio

COST = 80 ;   direttiva che assegna la costante
        LDA # COST ;   la costante è "parametrizzata"

Il 6502 manipola solo dati di 8 bit, ha un solo modo immediato su due byte. Il 6800 ha alcune volte bisogno di costanti su 16 bit: esiste quindi un secondo modo immediato, su tre byte di cui due sono occupati dalla costante.
Gli assembler simbolici associati al 6502 offrono una facilitazione interessante per manipolare una costante divisa in due: se TIZIO rappresenta un indirizzo (16 bit):

LDX   #<**TIZIO** carica in X il byte basso di TIZIO
LDY   #>**TIZIO** carica in Y il byte alto di TIZIO

Un altro modo che non è propriamente un modo di indirizzamento è il modo chiamato "implicito". Questo modo si applica alle istruzioni di manipolazione interna del microprocessore come scambio tra registri (esempio: TAX) o una azione su di un flag (esempio: SEC). Questo modo si applica anche alle istruzioni di manipolazione della pila o al ritorno da sottoprogrammi perché in quel caso, benché vi siano dei dati da cercare in memoria, l'indirizzo non deve essere fornito perché il microprocessore sa "implicitamente" ritrovarlo con l'aiuto del puntatore alla pila.
In assembler simbolico, le istruzioni implicite si presentano senza operando e si assemblano in un solo byte:

Esempio
60 RTS ; Ritorno da sottoprogramma

Si può ricollegare al modo implicito il modo che i costruttori chiamano *modo accumulatore*. Questo modo concerne le istruzioni unarie che possono agire su una cella di memoria o sull'accumulatore, come gli scorrimenti e le rotazioni

Esempio

0A   **ASL A**; *scorrimento dell'accumulatore*
06   10 **ASL M**; *scorrimento della cella di indirizzo 10*

Esiste un imperativo: **A** non può servire come nome di variabile, altrimenti ci sarebbe confusione per l'assembler simbolico; ed è così per tutti i nomi dei registri **A, X, Y, P, PC, S**: sono "riservati".

Veniamo ora ai modi di indirizzamento propriamente detti. Il più naturale consiste nel dare semplicemente l'indirizzo dell'operando voluto su due byte: così caricare in accumulatore il contenuto dell'indirizzo 1000 si codificherà su tre byte: il primo byte significherà "carica in accumulatore", poi due byte che conterranno 1000.
Questo si chiama *indirizzamento diretto*. Il 6502, come lo Z80, ha una particolarità su questo punto: i due byte che formano l'indirizzo sono invertiti, si trova prima il byte basso, poi il byte alto: così, LDA $1000 si assembla in:

indirizzo	n	n+1	n+2
	AD	00	01

Ugualmente, LDA 1000 (qui, abbiamo 1000 in decimale, cioè 3E8 in esadecimale) si assembla in **AD E8 03**.
Fra i microprocessori più utilizzati, solo il 6800 non fa questa inversione (di cui bisogna preoccuparsi solo facendo l'assemblaggio a mano). Questa inversione migliora l'efficacia dei calcoli di indirizzo, e permette di guadagnare dei cicli in certe istruzioni: così, l'istruzione di memorizzazione diretta dell'accumulatore A abbisogna di quattro cicli sul 6502 ma cinque sul 6800.
Il modo di *indirizzamento pagina zero* deriva dall'indirizzamento diretto, ma permette di guadagnare un byte di ingombro in memoria e un ciclo sulla durata di esecuzione.
Quando l'indirizzo cercato è inferiore a 256 (si dice che è nella pagina zero), si può benissimo impiegare l'indirizzo diretto; il terzo byte dell'istruzione sarà nullo (*).

Esempio

(LDA) AD zz 00. Ebbene il 6502 (come il 6800) permette di non scrivere il terzo byte, a condizione di anticipare il microprocessore modificando il codice operativo: A5 zz.
In questo momento l'istruzione occuperà due byte al posto di tre e la sua esecuzione durerà tre cicli invece di quattro. È per questo che è consigliabile implementare i dati più spesso utilizzati in pagina zero, ma si è limitati a 256 byte.
Si noterà che il byte di codice operativo è modificato: dipende dalla natura dell'operazione da effettuare, ma anche dal modo di indirizzamento.

(*) La Motorola chiama impropriamente questo modo "diretto".

Il modo che ora stiamo per vedere comporta l'addizione del contenuto di un registro alla parte di indirizzo PI che è presente nell'istruzione: si dice che si fabbrica un indirizzo effettivo IE:

IE = (registro) + PI

Con questa notazione, l'indirizzamento immediato si descriverebbe come operando = PI e l'indirizzamento diretto si descriverebbe come operando = (PI) ove, tradizionalmente, le parentesi significano "contenuto di". Si potrà verificare che la preoccupazione costante degli inventori che hanno introdotto tutta questa varietà di modi è stata l'economia di memoria e di tempo di esecuzione.

Indirizzamento relativo

È il modo di indirizzamento utilizzato per i salti condizionati del 6502. Si potrebbe avere, per questo un indirizzamento diretto ove PI sarebbe l'indirizzo di destinazione. È il caso dell'istruzione di salto incondizionato JMP.

JMP $1234 si assembla in **4C 34 12**

Come si potrebbe guadagnare un byte? La tecnica della pagina zero non può applicarsi alle istruzioni di salto perché bisogna poter saltare in qualsiasi punto della memoria e non soltanto in pagina zero. Ma ciò che si constata esaminando i programmi è che il 95% dei salti si fanno a piccola distanza dal punto di partenza.
È naturalmente il caso dei cicli, come abbiamo visto qui sopra: i cicli sono spesso corti. Da cui l'idea di codificare la distanza su un byte e non l'indirizzo stesso: l'indirizzo a cui saltare sarà dunque **IE** = indirizzo dove siamo + **PI**. Ma, per definizione, l'indirizzo dove siamo è il contenuto di PC. Da cui IE = (PC) + PI.
Il modo si chiama indirizzamento relativo a PC: altri sistemi possono avere un indirizzamento relativo ad altri registri, e PI si chiama spiazzamento (displacement). Poiché il salto deve poter essere in avanti o all'indietro, lo spiazzamento è con segno : è rappresentato in complemento a due e deve essere compreso tra –128 e +127.
Molto importante quando si esegue l'assemblaggio a mano: il valore di PC da prendere per calcolare lo spiazzamento è l'indirizzo dell'istruzione + 2 perché PC punta già verso l'istruzione seguente quando si esegue il salto.

Esempio

Se in 1000 (esa) abbiamo BNE all'indirizzo 1050, PC vale 1002 e lo spiazzamento è 1050–1002 = 4E da cui:

1000 D0 4E **BNE** $1050
1002 istruzione seguente

Se, in $340, abbiamo BPL a $33A, lo spiazzamento è: 33A–342=–(342–33A)=–(08)=(FF–08)+1=F7+1=F8: 0340 10 F8 **BPL $33A**.

Indirizzamento indicizzato

In questo sistema di modi, l'indirizzo effettivo si ottiene aggiungendo a **PI** il contenuto di uno dei registri indice X o Y. **PI** è su un byte (si dice che si ha l'indirizzamento pagina zero indicizzato) o su due byte (indirizzamento diretto indicizzato).
Questi modi sono utili per accedere a degli elementi di array; esattamente come in Basic si adoperano degli array indicizzati.
PI sarà l'indirizzo dell'array, mentre **X** svolgerà il ruolo di indice, caricato con il numero di elemento al quale si vuole accedere.

Esempio

Su un C64, il programma qui sotto sbianca le prime cinque linee dello schermo (indirizzo da 1024 a 1024+199).

```
10 FOR X = 0 to 199
20 POKE 1024 + X, 32 : REM 32 è il codice dello spazio
30 NEXT X
```

In assembler si scriverà:

```
C000  A2  00                LDX #0
C002  A9  20                LDA #32
C004  9D  00  04 CICLO      STA $400,X
C007  E8                    INX
C008  E0  C8                CPX #200
C00A  D0  F8                BNE CICLO
C00C  60                    RTS
```

Come abbiamo già notato, una versione con X decrementato sarebbe più efficiente.
Una limitazione dell'indirizzamento indicizzato del 6502 è che la grandezza del vettore deve essere inferiore a 256 elementi poiché X e Y hanno solo 8 bit.
È ampiamente sufficiente per la maggior parte delle applicazioni e la presenza di due registri indice aumenta la flessibilità.
Il 6800 ha un solo registro indice IX di 16 bit. In compenso, **PI** è limitato a 8 bit. Questo vuole dire che il vettore può avere una grandezza qualsiasi, ma il suo inizio deve essere in pagina zero. Infatti, è **PI** che conterrà l'indice e **IX** conterrà l'inizio del vettore. Di fatto non

è esattamente un indirizzamento indicizzato: si parla piuttosto di indirizzamento con registro modificatore di base.
Un'altra variante realizzata dal 6800 sarebbe l'indirizzamento paginato dove un registro di 8 bit contiene la parte alta dell'indirizzo, PI su 8 bit contiene la parte bassa: è analogo all'indirizzamento pagina zero, il registro serve a contenere un numero di pagina diverso da zero:

IE = (registro) **PI** (sono concatenati)

Indirizzamento indiretto

Veniamo ora a dei modi che solo il 6502 possiede sul mercato dei microprocessori a 8 bit. Questi modi sono implementati anche su alcuni minielaboratori.
Vi ricorderete la progressione che c'è fra l'indirizzamento immediato e l'indirizzamento diretto: nell'indirizzamento immediato, la parte indirizzo PI dell'istruzione è l'operando cercato, operando = PI. Nell'indirizzamento diretto, la parte indirizzo è l'indirizzo dell'operando:

Operando = (PI)

Ebbene, esiste un modo in cui PI è l'indirizzo dell'indirizzo dell'operando: operando = ((PI)).
PI (vedi **figura 4a**) punta ad un byte (11). Questo byte e il seguente (hh) formano l'indirizzo hh11 dell'operando cercato. Sul 6502, questo modo esiste solo per l'istruzione di salto incondizionato JMP.

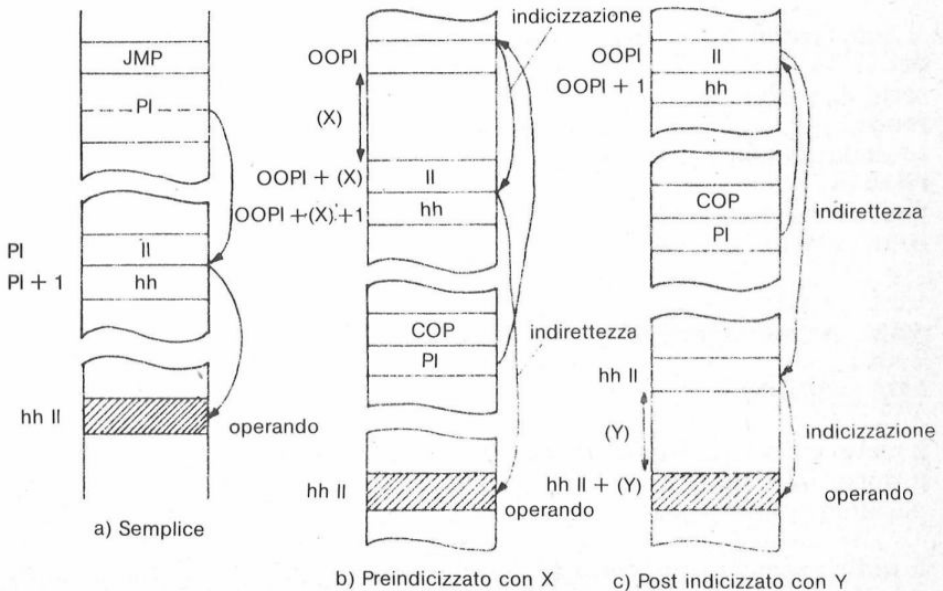



Fig. 4. Indirizzamento indiretto.

Questo è utilizzato, fra l'altro, per simulare in RAM i vettori di interruzione: si sa che quando arriva un'interruzione, il microprocessore prende, dagli indirizzi FFFE a FFFF, due byte che costituiscono l'indirizzo di inizio delle routine di risposta all'interruzione.
Ebbene, la prima istruzione della routine di interruzione e JMP (IVRAM) salto indiretto a IVRAM. Ma IVRAM è in RAM, ciò che permette, all'utente, di piazzarvi l'indirizzo della propria routine di risposta all'interruzione.
Questo si può fare sul 6800, in maniera più complessa:

```
LDX IVRAM ; carica il contenuto di IVRAM in IX
JMP 0,IX ; salta a 0 + contenuto di IX
```

Un'altra utilizzazione è di saltare ad un indirizzo che è il risultato di un calcolo.
Gli altri ***indirizzamenti indiretti*** del 6502 sono combinati con un'indicizzazione. La questione è di sapere se l'indicizzazione (aggiunta del registro indice all'indirizzo) ha luogo prima o dopo l'acquisizione del primo indirizzo.
Ebbene, il 6502 li offre entrambi, ma specializzando i suoi due registri indice: X per la preindicizzazione, Y per la postindicizzazione. I due modi così ottenuti sono descritti in **figura 4b** e **4c**.
Si vede che l'indirizzo di base è ristretto alla pagina zero. Ciò fa della pagina zero un blocco di 128 puntatori che hanno molte applicazioni.

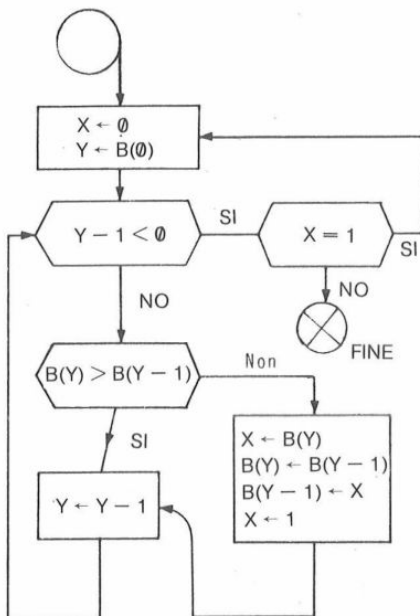
L'***indirizzamento indiretto preindicizzato con X*** permette di gestire dei vettori di puntatori. In particolare, può servire a trasmettere una serie di parametri fra sottoprogrammi: il programma chiamante memorizza gli indirizzi dei parametri a partire da un indirizzo convenuto della pagina zero, diciamo $50: avremo, dunque, in 50 e 51 l'indirizzo del primo argomento, in 52 e 53 quello del secondo, ecc.
Ora, se nel sottoprogramma vogliamo recuperare il parametro numero n, si farà:

```
LDA  #n
ASL  A; raddoppia A
TAX
LDA  ($4E,X)
```

n deve essere moltiplicato per due poiché ciascun parametro ha il proprio indirizzo su due byte. L'indirizzo di base è 4E (indirizzo del parametro 0).

L'***indirizzamento indiretto indicizzato con Y*** permette di manipolare degli array il cui indirizzo di inizio è indiretto, quindi può risultare da un calcolo. Diamo qui sotto un esempio.

Esempio completo

Diamo, per concludere, un esempio completo di programma di sorting (ordinamento), prima in Basic, poi in assembler 6502 (**figura 5**).
Si tratta di ordinare, in ordine crescente, gli elementi dell'array B. Viene utilizzato il metodo BUBBLE SORT (ordinamento a bolle); l'array è percorso confrontando gli elementi di coppie successive; se gli elementi di una coppia sono già ordinati, si passa alla coppia seguente; se non sono ordinati, sono scambiati di posizione; se il percorso è stato effettuato senza scambi significa che l'array è ordinato. Altrimenti viene iniziato un nuovo percorso. L'operazione è ripetuta sinché l'array diventi ordinato.
Il numero di un elemento all'interno dell'array è Y. Il registro X serve da variabile intermedia per gli scambi, e da indicatore di scambio: X=0 se non ci sono stati scambi, X=1 se ve ne sono stati.
Nell'elemento 0 dell'array, mettiamo la lunghezza utile (il numero di elementi). Per aumentare l'efficienza, l'array è percorso al contrario (da cui STEP –1 in Basic). Nella versione assembler, viene utilizzato l'indirizzamento indiretto. L'indirizzo dell'array da ordinare è nei byte 2 e 3 della memoria. È sufficiente completarlo con l'indirizzo dell'array da ordinare prima di chiamare il sottoprogramma di ordinamento.

FLOW CHART

```
10 DIM B(100)
20 B(0)=100
30 X=0
40 FOR Y=B(0) TO 2 STEP-1
50 IF B(Y)<=B(Y-1)
GOTO 100
60 X=B(Y)
70 B(Y)=B(Y-1)
80 B(Y-1)=X
90 X=1
100 NEXT Y
110 IF X<>0 GOTO 30
READY.
```

BASIC

```
0000               B = $2
0000               * = $C000
C000 A2 00  ORD    LDX #0     ;LUNGHEZZA ARRAY
C002 A1 02         LDA (B,X)  ;LUNGHEZZA ARRAY
C004 A8            TAY        ;IN Y
C005 B1 02  PAS    LDA (B),Y  ;ELEMENTO 2
C007 88            DEY        ;PUNTA VERSO ELEMENTO 1
C008 F0 12         BEQ FINE   ;FINE ARRAY?
C00A D1 02         CMP (B),Y  ;CONFRONTA CON ELEMENTO 1
C00C B0 F7         BCS PAS    ;CICLO SE NON SCAMBI
C00E AA     SCAM   TAX        ;ELEMENTO 2 IN X
C00F B1 02         LDA (B),Y  ;ELEMENTO 1 IN A
C011 C8            INY        ;PUNTA ALL'ELEMENTO 2
C012 91 02         STA (B),Y  ;ELEMENTO 2 VA IN 1
C014 8A            TXA        ;ELEMENTO 2
C015 88            DEY        ;PUNTA ALL'ELEMENTO 1
C016 91 02         STA (B),Y  ;ELEMENTO 1 VA IN 2
C018 A2 01         LDX #1     ;SCAMBIO AVVENUTO
C01A D0 E9         BNE PAS    ;PROSSIMA COPPIA
C01C 8A     FINE   TXA        ;GUARDA SE X =0
C01D D0 E1         BNE ORD    ;ALTRIMENTI FAI RICOMINCIA
C01F 60            RTS        ;RITORNO
```

Fig. 5. Programma di ordinamento.

Capitolo 2

Nozioni preliminari

Prima di proseguire, bisogna vedere due elementi fondamentali.
Per prima cosa, in linguaggio macchina, si agisce sui dati al livello più elementare. Bisogna dunque comprendere in dettaglio come sono memorizzati e come sono codificati: è il problema della rappresentazione dei dati. Inoltre, non è possibile apprendere ragionevolmente il linguaggio assembler (non più di tutti gli altri linguaggi, d'altronde) senza esercizi e impegno.
Bisogna dunque provare dei programmi. Per fare ciò, bisogna introdurli in memoria ed eseguirli.
Queste operazioni sono immediate ed automatiche in Basic; ma, per il linguaggio macchina, bisogna disporre di un programma speciale, il monitor. Sarà descritto in questo capitolo poiché ne avremo bisogno in quelli successivi.

RAPPRESENTAZIONE DEI DATI

Allo stato attuale della tecnologia tutti i dati maneggiati dai calcolatori, qualsiasi essi siano, sono memorizzati in forma binaria, cioè sotto forma di una serie di zero ed uno, o come insieme di oggetti fisici a due stati: la tensione elettrica a 0 V oppure a 5 V, presenza o assenza di un foro su una scheda, magnetizzazione in un senso o nell'altro. La lotta contro i disturbi è molto più facile con un elemento che ha solo due stati possibili che con un elemento suscettibile di assumere una moltitudine di configurazioni.
Ogni elemento che può assumere due configurazioni ed è dunque capace di contenere la risposta sì o no ad una domanda si chiama BIT (Binary digIT=cifra binaria). Indicheremo gli stati con 0 e 1.

Ma, più spesso, sono dei gruppi di bit ad essere manipolati. Il raggruppamento più usuale, entità manipolata dal C64, è l'insieme di 8 bit, chiamato BYTE.
Un byte, come si trova in memoria, può rappresentare delle informazioni di natura molto differente ed è il trattamento che il programma riserverà al byte che indicherà che tipo di informazione è presa in considerazione.
Un byte può rappresentare:

1. un numero, con segno o no, eventualmente in Binario Codificato Decimale;
2. un carattere alfanumerico.

Può anche essere un elemento di un'informazione composta, codificata su più byte:

3. parte di un numero in precisione multipla, intero o reale;
4. un indirizzo;
5. un codice operativo;
6. un carattere di una stringa.

Numeri interi senza segno

È la rappresentazione più naturale: il numero è scritto nel sistema binario, ogni cifra 0 o 1 corrisponde ad un bit.
Sappiamo che un numero è rappresentabile in qualsiasi base di numerazione. Ogni sistema è caratterizzato dalla base b e da una serie di b segni che rappresentano le cifre.
Così, nel sistema decimale, b=10 e le 10 cifre sono: 0 1 2 ... 9
Il numero 1369 vale:

$$9 + 6 \times 10^1 + 3 \times 10^2 + 1 \times 10^3$$

Questa formula è valida qualunque sia la base: le cifre hanno per valore rispettivo 0 1 2 ... (b–1) e il numero $a_p\, a_{p-1}...a_2a_1a_0$ vale:

$$\sum_{0}^{p}{}_i \; a_ib^i = a_0 + a_1 \,.\, b + a_2b^2 + ... + a_pb^p$$

Nel sistema binario, b=2 e le cifre sono 0 e 1. Il byte $a_7a_6...a_1a_0$ rappresenta il valore:

$$a_0 + a_1 \times 2 + a_2 \times 2^2 + ... + a_7 \times 2^7$$

Il numero più piccolo possibile è 0 (tutte cifre nulle), il numero più grande che possa essere rappresentato con un byte in cui tutte le cifre valgano 1 è $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$. Il numero di

combinazioni binarie possibili, compreso lo zero, è 256, che corrisponde ovviamente a 2^8.

Notazione esadecimale

I numeri binari sono difficoltosi da maneggiare poiché hanno molte cifre, anche per piccoli valori. Ma il valore di un byte può scriversi:

$$a_0 + a_1 \cdot 2 + \dots + a_7 \cdot 2^7 =$$

$$= \underbrace{a_0 + a_1 \cdot 2 + a_2 2^2 + a_3 2^3}_{A_0} + 2^4 \underbrace{(a_4 + a_5 \cdot 2 + a_6 \cdot 2^2 + a_7 \cdot 2^3)}_{A_1}$$

$$= A_0 + A_1 \times 16$$

A_0 e A_1 valgono al massimo 15 $(1 + 2 + 2^2 + 2^3)$. Sono, quindi, le cifre della rappresentazione del numero in base 16.
D'ora in poi indicheremo sempre i numeri in esadecimale, ma si tratta unicamente di una notazione: all'interno della macchina, i dati sono rappresentati in binario nei byte e immaginiamo il contenuto binario dei byte in esadecimale sui nostri fogli.
Il passaggio da binario a esadecimale e viceversa è immediato, con l'aiuto della **tabella 2.1.** Per indicare le ultime cifre esadecimali, si usano le lettere dalla A alla F. Per passare dal binario all'esadecimale, si raggruppano i bit 4 a 4 (si completano ove occorra con degli 0 a sinistra) e si sostituisce ogni quartetto con la cifra esadecimale equivalente; dall'esadecimale al binario, si sostituisce ogni cifra con i suoi quattro bit equivalenti.

Tab. 2.1. **Cifre esadecimali.**

Esadecimale	Binario	Decimale	Esadecimale	Binario	Decimale
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

Per passare dal binario al decimale e viceversa, si passa per l'intermediario dell'esadecimale. Per la conversione esadecimale-decimale, si sostituisce ogni termine dell'espressione $A_0 + A_1 \cdot 16 + \dots$ col suo valore.

Esempio

Sia da convertire 11101011 in decimale. Il numero è EB in esadecimale, quindi: B + E · 16 = 11 + 224 = 235.
La **tabella 2.2.** fornisce, in funzione di x e di n, i valori di $x \cdot 16^n$.

Tab. 2.2. **Valori di $x \cdot 16^n$.**

X/N	3	2	1	0	X/N		3	2	1	0
1	4096	256	16	1	9		36864	2304	144	9
2	8192	512	32	2	10	A	40960	2560	160	10
3	12288	768	48	3	11	B	45056	2816	176	11
4	16384	1024	64	4	12	C	49152	3072	192	12
5	20480	1280	80	5	13	D	53248	3328	208	13
6	24576	1536	96	6	14	E	57344	3584	224	14
7	28672	1792	112	7	15	F	61440	3840	240	15
8	32768	2048	128	8	16	10	65536	4096	256	16

Ovviamente, è il C64 (e la sua stampante) che ci·ha fornito questa tabella. Può essere utilizzata anche per la conversione decimale → esadecimale.

Esempio

Per convertire 4238 in binario, si cerca il numero più grande presente nella tabella e < 4238.
È 4096, da cui: 4238 = 4096 + 142 = 142 + 1000 esa.
Allo stesso modo: 142 = 128 + 14 = 14 + 80 esa = E + 80 da cui 4238 = 108E esa = 0001 0000 1000 1110

Esercizio 2.1 *Scrivere un programma in Basic che converte da decimale ad esadecimale e viceversa.*

Aritmetica binaria ed esadecimale

Le regole delle operazioni sono generali, indipendenti dalla base di numerazione in questione. Quindi, si possono applicare, in binario od in esadecimale, le stesse regole conosciute per il sistema decimale.

Esempio

Addizione. Il riporto si fa a due in binario, ed a 16 in esadecimale.
È la sola differenza:

Decimale	*Esadecimale*	*Binario*
14	OE	00001110
+ 23	+ 17	00010111
37	25	00100101
	14 + 7 = 21 = 15esa scrivo 5 e riporto 1	1 + 0 = 1; 1 + 1 = 2 (10): scrivo 0 e riporto 1 1 + 1 + 1 = 3 (11): scrivo 1 e riporto 1, ecc.

Esempio

Moltiplicazione. Uno scorrimento a sinistra moltiplica per 10 in decimale, per 16 in esadecimale, per 2 in binario:

```
   3      11
×  5  ×  101
  15      11
        11××
        1111
```

Il prodotto di un numero di p bit per un numero di n bit ha al massimo n + p bit: 2 byte danno un prodotto di 16 bit.

Numeri interi con segno

Sappiamo ora rappresentare in binario i numeri interi positivi compresi tra 0 e 255. Si pongono però due problemi:

- la limitazione sulla grandezza del numero (la soluzione sarà di prendere una rappresentazione di più byte);
- la restrizione ai numeri positivi.

Per rappresentare i numeri, di qualsiasi segno, è sufficiente sacrificare un bit per rappresentare il segno. Tradizionalmente si utilizza il bit più significativo (bit 7, il più a sinistra) per rappresentare il segno: si crea la convenzione 0 = numero positivo, 1 = numero negativo. Ciò permette di avere, per i numeri positivi, esattamente la stessa rappresentazione precedente, salvo che si hanno solo 7 bit "utili" (il bit 7 vale 0) e quindi i numeri sono compresi fra 0 e 127 (7F esa).
Per i numeri negativi sono possibili tre rappresentazioni:

Segno-valore

Il bit più a sinistra è il bit di segno, gli altri rappresentano il valore assoluto:

+ 3 : 00000011; –3 : 10000011

L'inconveniente è che le regole delle operazioni non sono conservate; occorrono delle regole differenti a seconda del segno degli operandi. Per esempio, se cerchiamo di fare +3+(–3) con le regole normali, otterremo 10000110, che non è corretto (dovremmo ottenere 00000000).

Complemento a 1

Per avere l'opposto di un numero si complementano tutti i bit (1 → 0; 0 → 1). Ma, ancora, le regole di operazione non sono conservate:

+ 3 : 00000011 ; –3 : 11111100 ; + 3 +(–3) = 11111111

Complemento a 2

Per avere l'opposto di un numero, si calcola il complemento a 1, poi si aggiunge 1 tralasciando l'eventuale riporto al di là del bit di segno.

Esempio

+ 3 : 00000011 ; –3 : 11111100 + 1 = 11111101 (FD)

In esadecimale, si sottrae da FF il numero di cui si vuole l'opposto, poi si aggiunge 1.

Esempio

Per calcolare –3: FF–3 = FC; FC + 1 = FD.

Esercizio 2.2 *Di quale numero è l'opposto FF?*

Prendiamo l'opposto di FF. FF–FF = 00; 00 + 1; FF è l'opposto di 1, dunque FF (11111111) è la rappresentazione di –1.
È la notazione del complemento a 2 che è maggiormente implementata sui calcolatori. In particolare è impiegata anche sul 6502.
Infatti con questa rappresentazione si possono applicare le regole abituali dell'addizione e della sottrazione senza occuparsi del segno degli operandi. Ciò semplifica la realizzazione dell'unità aritmetico-logica all'interno del microprocessore.
Mostriamo qualche esempio di operazione. I numeri positivi possibili sono:

00000000 (0 = 0), 00000001 (01 = 1) ... 01111110 (7E = +126),
01111111 (7F = +127)

Il numero 10000000 (80) non è l'opposto di nessun numero positivo. È negativo poiché ha il bit 7 a 1, e si ottiene facendo –127 –1, è –128. Esiste un numero negativo in più dei numeri positivi perché 0 è positivo dal punto di vista del suo bit di segno.

Addizione

```
2 + 1  → 0000 0010        2 + (− 1) → 0000 0010          02
       + 0000 0001                  + 1111 1111          FF
       = 0000 0011  = 3           [1] 0000 0001   o  [1] 01
                                   ↓
                                    Riporto tralasciato:
                                    il risultato è 1

2 + − 3 → 0000 0010           02          − 3 + 1    FD
        + 1111 1101        +  FD                   + 01
        = 1111 1111 = − 1     FF                     FE = − 2

− 1 + (− 2) → 1111 1111       FF
            + 1111 1110    +  FE
          [1] 1111 1101   [1] FD = − 3
           ↓
          Riporto tralasciato
```

Overflow

Alcune operazioni sono molto bizzarre.

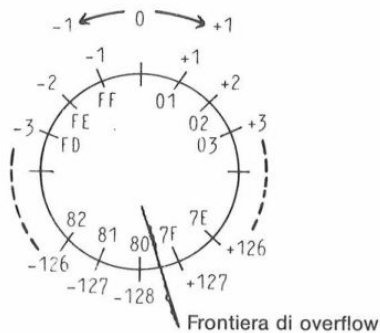
Esempio

```
  64 →  40         0100 0000
+ 64 +  40         0100 0000
        80  [1]     000 0000
             ↓ Negativo
```

La somma di due numeri positivi (2 × 64) è un numero negativo (− 128)! Allo stesso modo:

```
  − 127  →  81        1000 0001
+ −   2     FE        1111 1110
            7F    [1] 0111 1111
                      ↑
                      Positivo
```

Il fenomeno è dovuto al limitato numero di bit disponibili: quando un riporto passa dal bit 6 al bit 7, il risultato non è più corretto perché il bit 7 è il bit di segno. Se si disponesse di un bit in più il numero resterebbe positivo e il risultato sarebbe corretto.

La rappresentazione ha una caratteristica limitata tale da apparire come ciclica:

Propagazione del segno

Un'ultima proprietà interessante di questa rappresentazione è che, per passare da una notazione ad un'altra con più bit, basta ripetere il bit di segno sulla sinistra.

Esempio

5 su 8 bit: 0000 0101; su 16 bit 0000 0000 0000 0101 (0005)
–1 su 8 bit: 1111 1111 (FF); su 16 bit 1111 1111 1111 1111 (FFFF)

Decimale codificato binario

Un'altra maniera di rappresentare i numeri sarebbe di partire dalla notazione decimale e di codificare in binario ciascuna delle cifre, da cui il nome della rappresentazione. Ma quanti bit occorrono per rappresentare ogni cifra decimale?

Esercizio 2.3 *Sono sufficienti tre bit?*

Si utilizzano 4 bit per ciascuna cifra decimale. Ma, in 4 bit, si possono avere le cifre 0 1 2 3 4 5 6 7 8 9 A B C D E F. Solamente quelle dallo 0 al 9 sono utilizzate, ci sono 6 combinazioni inutilizzate che ci porranno dei problemi per i calcoli in modo decimale.
Un byte può valere da 00 a 99. Per andare più avanti, occorrono più byte. Anche qui, si può codificare il segno del numero o impiegare una notazione complementata.

Esercizio 2.4 *33 è un numero BCD. Da quale byte è rappresentato in esadecimale?*

Rappresentazione dei caratteri

Si decide che un carattere sarà rappresentato da un byte. Una stringa di caratteri sarà la giustapposizione di byte rappresentanti caratteri successivi. Ogni byte può rappresentare 256 caratteri differenti, ed è un set più che sufficiente.
Il C64 utilizza il set di caratteri ASCII. D'altra parte, un codice abbreviato è utilizzato dalla memoria dello schermo, il codice schermo.
I codici sono riuniti in esadecimale nella **tabella 2.3** a pagina 42.

Dati composti

Un byte da solo può costituire un dato. Abbiamo visto che, in questo caso, può rappresentare un numero intero compreso tra 0 e 255, oppure tra –128 e 127, o anche un carattere alfanumerico.
Ma un byte può anche essere membro di un gruppo di byte che rappresentano un dato: infatti alcuni dati hanno bisogno di più byte per essere rappresentati, per esempio i numeri interi più grandi di 255. Chiameremo questi dati: dati composti.
I diversi byte che formano un dato composto possono svolgere il medesimo ruolo (esempio: stringa di caratteri) o ruoli differenti (esempio: numero reale ove si distingue la mantissa dalla caratteristica). Non confondete tale affermazione con la nozione di dato strutturato (esempio: array, lista) ove sono raggruppati degli elementi che possono loro stessi essere formati da più byte.

Numeri in precisione multipla

Per rappresentare numeri più grandi di 255, bisogna utilizzare più byte. Per esempio, una coppia di byte rappresenta un numero di 16 bit, quindi compreso tra 0 e 65535. Se il numero è considerato con segno, rappresentato in complemento a 2, è compreso tra –32768 e +32767.
È il caso delle variabili con prefisso % nel Basic del C64.
L'ordine byte alto-byte basso nel quale sono memorizzati i due byte è poco importante (a condizione di conoscerlo e di attenervisi). Notiamo che, quando il numero rappresenta un indirizzo, per seguire la convenzione abituale del 6502, il byte basso precede il byte alto.

Esercizio 2.5 *Fare il complemento a 2 di* |*35*|*4A*|
basso alto

Numeri in virgola mobile

Si tratta di rappresentare i numeri reali, non interi. Ci si ispira alla notazione "scientifica" dei numeri. Per esempio, 1.233.000.000 si potrà scrivere $0{,}1233 \times 10^{10}$.

Tab. 2.3. **Codici caratteri del C64 (in esadecimale).**

CARATTERE	CODICE SCHERMO*	ASCII	CARATTERE	CODICE SCHERMO*	ASCII	CARATTERE	CODICE SCHERMO*	ASCII
A a	1	41	(	28	28 +		66	A6 +
B b	2	42	)	29	29 +		68	A8 +
C c	3	43	"	22	22 +		5C	DC
D d	4	44	'	27	27 +		77	B7 +
E e	5	45	#	23	23 +		78	B8 +
F f	6	46	$	24	24 +		62	A2 +
G g	7	47	%	25	25 +		79	B9 +
H h	8	48	&	26	26 +		6F	AF +
I i	9	49	£	1C	5C		72	B2 +
J j	A	4A	←	1F	5F		73	B3 +
K k	B	4B	π	5E	DE		6B	AB +
L l	C	4C	[	1B	5B		71	B1 +
M m	D	4D	]	1D	5D			
N n	E	4E	@	Ø	4Ø			
O o	F	4F	♣ a	41	C1	J	4A	CA
P p	1Ø	5Ø	♥ s	53	D3	K	4B	CB
Q q	11	51	♦ z	5A	DA	U	55	D5
R r	12	52	♣ x	58	D8	I	49	C9
S s	13	53		7D	BD +		63	A3 +
T t	14	54		6D	AD +	E	45	C5
U u	15	55		7Ø	BØ +	D	44	C4
V v	16	56		6E	AE +	C	43	C3
W w	17	57		7E	BE +		4Ø	CØ
X x	18	58		7C	BC +	F	46	C6
Y y	19	59		6C	AC +	R	52	D2
Z z	1A	5A		7B	BB +		64	A4 +
Ø	3Ø	3Ø +	spazio Sp	2Ø/6Ø	2Ø +		65	A5 +
1	31	31 +	O	4F	CF	T	54	D4
2	32	32 +	P	5Ø	DØ	G	47	C7
3	33	33 +	√	7A	BA +	B	42	C2
4	34	34 +	L	4C	CC		5D	DD
5	35	35 +	N	4E	CE	H	48	C8
6	36	36 +	M	4D	CE	Y	59	D9
7	37	37 +	V	56	D6		67	A7 +
8	38	38 +		5B	DB			
9	39	39 +						
+	2B	2B +		7F	BF	Return		ØD
–	2D	2D +	W	57	D7	Return		8D
*	2A	2A +	Q	51	D1	d		1D
/	2F	2F +		74	B4 +	q		9D
↑	1E	5E		75	B5 +	b		11
.	2E	2E +		61	A1 +	h		91
=	3D	3D +		76	B6 +	Home		13
<	3C	3C +		6A	AA +	Clr		93
>	3E	3E +		69	A9 +	Del		14
!	21	21 +		5F	DF	Inst		94
?	3F	3F +				RvsC		12
:	3A	3A +				OffC		92
;	3B	3B +						
,	2C	2C +						

Note alla tabella 2.3

Se il codice ASCII del carattere *x* è uguale ad *a*, allora il codice ASCII di *'shift' x* è uguale ad *a+128* (a + 80esa)
+ Quando un codice ASCII, a, è seguito dal segno +, si può anche usare il codice a+64 (a+40esa).
* Per ottenere il carattere in campo inverso, aggiungere 128 (80esa) al codice schermo.
Se vi sono due caratteri nella medesima colonna, il primo corrisponde al modo "grafico", il secondo al modo "minuscolo".

Tutti i numeri andranno rappresentati seguendo tale modello, ma poiché siamo in binario, ogni numero sarà nella forma:

$0{,}m \times 2^c$
ove m è la mantissa e c la caratteristica.

Ci basta, ora, rappresentare c ed m su un certo numero di byte, senza dimenticare il segno del numero e il segno della caratteristica (per i numeri piccoli, abbiamo le potenze negative di due).
Il C64 segue i principi di cui sopra per memorizzare i numeri reali ma, ed è per questo che la notazione si chiama "in virgola mobile", il numero è memorizzato sempre in forma "normalizzata": gli scorrimenti voluti sono effettuati affinché il bit più significativo della mantissa sia 1 e, naturalmente, l'esponente è manipolato di conseguenza.

Esempio

$1/4 = 0{,}01 \times 2^0$ (c = 0; m = 0100...)

Sarà memorizzato nella forma $0{,}1 \times 2^{-1}$ (c = –1; m = 1000...)
Il C64 rappresenta i numeri reali su 5 byte, di indirizzo da n ad n + 4.
All'indirizzo n, si trova la caratteristica, non in complemento a due, ma nella forma 80esa+esponente.
Per esempio, se l'esponente è 2, avremo 82, se è –2, avremo 7E. Quando la caratteristica è FF, l'esponente vale +127. Quando la caratteristica è 00, l'esponente vale –128: in effetti tutti i numeri che hanno tale caratteristica vengono considerati nulli.
La mantissa occupa i byte da n+1 ad n+4, cominciando dal byte più significativo (n+1). Il byte meno significativo è n+4.
Come si memorizza il segno del numero? Per questo si ricorda che il bit 7 del byte più significativo (n+1) è sempre a 1; è la definizione di normalizzazione.
Non è dunque necessario memorizzarlo; per convenzione, è sempre 1: allora, è rimpiazzato dal bit di segno: 1 se il numero è negativo, 0 se è positivo.

Esempio

Il numero + 1 $1 = 0,1 \times 2^1$ (poiché $0,1 = 1/2$)
da cui l'esponente 1, mantissa 1000... 0000
quindi

81	80	00	00	00

n

ma il bit di segno deve essere a 0 perciò

81	00	00	00	00

Il numero –1
stessa cosa del numero +1 ma il bit di segno deve essere a 1, perciò

81	80	00	00	00

Il numero +4
$4=0,1 \times 2^3$
da cui   83 80 00 00 00

e infine 83 00 00 00 00   per annullare il bit di segno

Il numero –0,25

$0,25 = 0,1 \times 2^{-1}$

da cui

7F	80	00	00	00

il bit di segno è conservato.

Il numero 5,5

$5,5 = 101,1 = 0,1011 \times 2^3$

perciò 83 B0 00 00 00 e infine 83 30 00 00 00

Il numero –100000

$100000 = 6250 \times 16$
$6250 = 4096 + 2048 + 96 + 10$
$= 1000 + 800 + 60 + A$ esa $= 186A0$ esa

perciò

$100000 = 186A0$ esa $= 1\ 1000\ 0110\ 1010\ 0000$ bin
$= 0,1\ 100\ 0011\ 0101\ 0000\ 0 \times 2^{17}$
$= 91\ C3\ 50\ 00\ 00$

La rappresentazione definitiva è quindi 91 C3 50 00 00 poiché il bit di segno è mantenuto a 1 (volevamo –100000).

Esercizio 2.6
a) *Spiegare la presenza del numero 91 nell'esempio precedente.*
b) *Rappresentare +100000.*

Per decodificare un numero, le operazioni sono esattamente le stesse, in ordine inverso.

Esempio

Cosa rappresenta FF D2 C4 31 BA?
L'esponente è FF–80 = 7F = +127.
Il numero è negativo poiché il bit 7 di D2 è a 1.
La mantissa è m=D2 C4 31 BA, che rappresenta

$0{,}m \times 2^{127}$
$0{,}m = m \times 2^{-32}$

quindi il numero è:

$-m \times 2^{127-32} = -m \times 2^{95}$
m vale $3{,}53607315 \times 10^9$ (grazie al programma dell'esercizio 2.1).
$2^{95} = 10^{95 \log 2} = 10^{95 \times 0{,}30103...} = 3{,}96140813 \times 10^{28}$
da cui il numero = $-1{,}40078289 \times 10^{38}$.

Esercizio 2.7 *Cosa rappresenta 90 25 5B 3A 00?*
In effetti, una variabile reale Basic è rappresentata nella memoria delle variabili nella forma

N	N	C	M	M	M	M

dove C ed MMMM sono i 5 byte visti precedentemente mentre | N | N |

sono i due byte che contengono le prime due lettere del nome della variabile in codice ASCII. È la ragione per la quale solo le prime due lettere della variabile sono significative in Basic.
Esiste una terza rappresentazione delle variabili reali nel C64. È quella degli accumulatori mobili. Il C64 ha tre zone di memoria in cui un numero in virgola mobile può essere memorizzato per i calcoli. Il principale è all'indirizzo n = 61, gli altri sono in 69 e 57.
L'accumulatore mobile è nella forma:

n ↓
C M1 M M M M2 S

ove C è la caratteristica, M1MMM è la mantissa, ma in M1 il bit 7 è messo a 1, M2 è una copia di M1 ma con il bit di segno come nella

rappresentazione precedente e S è un byte di codifica del segno: FF = negativo, 00 = positivo. M2 serve da zona ausiliaria per gli arrotondamenti.
Così, il numero +4 ha, per rappresentazione classica, 83 00 00 00 00 e, per rappresentazione in accumulatore: 83 80 00 00 00 00 00, mentre –4 ha per rappresentazione in accumulatore: 83 80 00 00 00 80 FF.

Esercizio 2.8 *Rappresentazione in accumulatore dei numeri +5,5 e –5,5?*

Ciò che è importante in questa rappresentazione è che si possono dedurre immediatamente due cose: l'ordine di grandezza limite dei numeri maneggiabili e la precisione della rappresentazione.
L'ordine di grandezza risulta dallo spazio di memoria limitato della caratteristica: l'esponente più grande è +127 ($2^{127} = 10^{37}$ circa). Dal lato dei numeri più piccoli, in modulo, l'esponente minore è —128 ($2^{-128} = 10^{-38}$ circa): tutti i numeri più piccoli saranno confusi con lo zero.
Per ciò che concerne la precisione, i due numeri più vicini che possono essere distinti differiscono per il bit 32 della mantissa. Questo numero vale $2^{-32} = 10^{-9,63}$. ciò spiega perché il C64 ha 9 cifre significative per i numeri reali.

Esercizio 2.9 *Abbozzate una rappresentazione che disponga di 14 cifre significative. Stessa domanda con 16.*

Rappresentazione di un indirizzo

Tutti gli indirizzi da maneggiare nel 6502 sono di 16 bit poiché tale microprocessore può indirizzare 64 K.
Occorrono due byte e la particolarità è che si trova sempre prima il byte basso e poi il byte alto.
Ciò si ritrova nella parte indirizzo delle istruzioni in linguaggio macchina che fanno riferimento ad un indirizzo di 16 bit (esempio: indirizzamento diretto), nei puntatori di indirizzamento indiretto, nei vettori di interruzione e in tutti i puntatori manipolati dal sistema operativo del C64.

Esempio

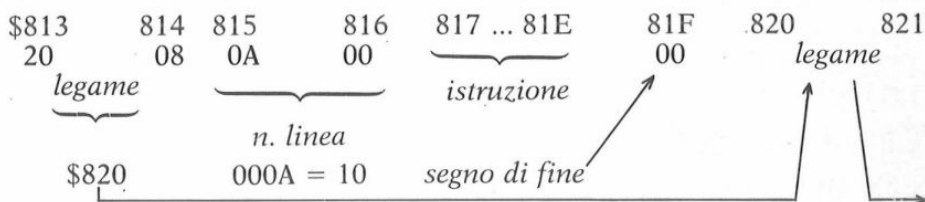
Gli indirizzi $37, $38 contengono l'indirizzo limite (+1) della RAM disponibile.
Su un C64, si ha allora $37 $38 poiché la ROM comincia in A000 esa.
00 A0

Rappresentazione di istruzioni

Le istruzioni-macchina sono rappresentate, come abbiamo visto al capitolo 1, su più byte.
Il primo (qualche volta unico) byte è una codifica arbitraria della natura dell'operazione e del modo di indirizzamento. Questi codici sono indicati in appendice. I byte seguenti codificano l'indirizzo di memoria relativo all'istruzione, tenuto conto del modo di indirizzamento utilizzato.
Allo stesso modo, esiste un metodo di codificare in memoria le istruzioni di un programma Basic. Lo descriveremo a titolo indicativo.
Infatti, un'istruzione Basic è rappresentata, all'ingrosso, come la stringa di caratteri che la compongono.
Ci sono solo due eccezioni a tale schema:

1. Le parole-chiave Basic sono rappresentate da un byte che costituisce un codice operativo. Ciò spiega due cose: non si guadagna niente in spazio di memoria ad abbreviare le parole-chiave e se lo si è fatto il listing restituisce le parole-chiave non abbreviate.
2. La linea di istruzioni termina con un byte 00 (segno di fine) e comincia con 4 byte particolari: i primi due formano un puntatore al primo byte della linea di istruzioni seguente. Lo si chiama legame. Gli altri due codificano il numero della linea corrente.

Esempio

In \$813 comincia la linea 10. La linea seguente in \$820. Si troverà in memoria:

Dopo l'ultima istruzione del programma si trova un'istruzione a legame nullo (e null'altro): 00 00. Quindi la fine del programma è segnalata dalla presenza di tre byte nulli consecutivi.

Stringhe di caratteri

Una stringa di caratteri è rappresentata semplicemente dal seguito di byte che codificano ciascun carattere.
Ciò è, infatti, molto potente: purché si abbia sufficiente memoria – e dischi di grande capacità esistono – si può far entrare, in un elaboratore, qualsiasi informazione che può pronunciarsi o scriversi sia l'enciclopedia Treccani che le opere complete di Dante!

Questa o quella descrizione di Dante, talmente approfondita che sia, è accessibile all'elaboratore per il quale è semplicemente una stringa di caratteri.
Secondo i casi, i caratteri successivi possono essere memorizzati salendo o discendendo la memoria.
In Basic, le variabili stringhe sono memorizzate in due volte: la stringa propriamente detta dall'indirizzo **a** all'indirizzo a + *l* –1 (*l* = lunghezza stringa) e un descrittore di 7 byte nella forma N1 N2 L A1 A2 00 00. N1 è il primo carattere del nome; N2 è il secondo carattere +80 esa; L è la lunghezza. A1 A2 formano un puntatore verso l'indirizzo **a** di inizio della stringa.

Esercizio 2.10 *Qual è la conseguenza essenziale di questa rappresentazione?*

Si vede, con i descrittori di variabile, un eccellente esempio di dato strutturato in cui ogni byte svolge un determinato ruolo.
Un altro esempio è quello delle istruzioni Basic in cui si implementa un concatenamento.
Gli array semplici formano una struttura meno complicata poiché i byte hanno il medesimo ruolo, sono solamente accostati. Nondimeno, alcune volte si implementano gli array con al primo byte la lunghezza seguita dai byte di dati propriamente detti.

TAB | LUNGHEZZA | DATO 1 | DATO 2 | ... | DATO N |

L'array è allora indipendente. Se AR è il suo indirizzo di base, si accede al byte **p** facendo:

```
LDX #p
LDA AR,X
```

e il test finale sarà

```
CPX AR
BNE INIZIO
```

IL MONITOR ESADECIMALE

Il monitor esadecimale è un programma capace:

- di visualizzare in esadecimale il contenuto di ogni cella di memoria;
- di modificarne il contenuto;
- di visualizzare e di modificare il contenuto dei registri interni del 6502;
- di eseguire codice in linguaggio macchina a partire da un indirizzo specificato dall'utente.

Questo monitor è il minimo di cui bisogna disporre per poter editare, eseguire e mettere a punto un programma in linguaggio macchina.
È il software di base di cui si dispone sui microelaboratori ad una sola scheda quali il KIM.
L'inconveniente sta nella scomodità di dover tradurre in esadecimale il programma scritto in assembler simbolico. Questo assemblaggio a mano può essere fatto solo per programmi brevissimi.
Perciò lo utilizzeremo per programmi estremamente semplici che scriveremo al capitolo 3. Di seguito, descriveremo i monitor di questo genere di cui si dispone per il C64.
Ci sono, infatti, più monitor in linguaggio macchina disponibili per il C64.

1. *SUPERMON 64* di J. Butterfield che è stato pubblicato dalla rivista americana "Compute" e che è di dominio pubblico, quindi utilizzabile gratuitamente.
2. *MONITOR 8000 o C000*. È il monitor presente sul dischetto assembler venduto da PROCEP. Ha due versioni che si caricano in $8000 o in $C000, a seconda dell'utilizzazione della memoria che desiderate.
3. *La cartuccia monitor della AUDIOGENIC*. Il suo vantaggio è di essere in cartuccia, dunque sempre pronta all'uso.

Diamo, qui, delle indicazioni sull'uso dei tre; le loro funzioni sono praticamente identiche.

- SUPERMON si carica facendo LOAD e poi RUN;
- MONITOR, facendo LOAD "...", 1,1 poi SYS 8*4096 oppure SYS 12*4096 secondo la versione;
- AUDIOGENIC si attiva facendo SYS 8*4096 o SYS 32768.

In ogni caso si ottiene una visualizzazione del tipo:

```
B*
   PC  SR AC XR YR SP
.;803E 32 00 83 00 F6
```

Il punto segna l'attesa di un comando. La visualizzazione è quella dello stato dei registri: PC program counter (contatore di programma), SR status register (registro di stato), AC accumulatore, X, Y, SP stack pointer (puntatore alla pila).
I principali comandi del monitor sono: **M** visualizzazione memoria, **R** visualizzazione registri, **G** esecuzione, X ritorno al Basic, **L** caricamento programma, **S** scaricamento programma su disco o nastro.

Comando M

Questo comando permette di visualizzare una zona di memoria ed, eventualmente, di modificarla.

Il comando si scrive: .M ␣ ind1,ind2

ove **ind1** è l'indirizzo di partenza ed **ind2** l'indirizzo di fine della zona da visualizzare.
Gli indirizzi vanno scritti in esadecimale su quattro cifre (A0 va scritto 00A0). Lo spazio tra **M** e gli indirizzi è obbligatorio.
La virgola che separa gli indirizzi può essere sostituita da uno spazio o un trattino.
La visualizzazione può superare ind2 se ciò è necessario per completare una linea.

Esempio

Premete 'clr'. Poi (in alto sullo schermo), premete M 0400 - 0420.
Si otterrà:

```
        M  0400-0420
.:0400 0D 20 30 34 30 30 2D 30
.:0408 34 32 30 2E 20 20 20 20
.:0410 20 20 20 20 20 20 20 20
.:0418 20 20 20 20 20 20 20 20
.:0420 20 20 20 20 20 20 20 20
     .■
```

0400 è la memoria dello schermo, si riconoscono in 0400 0D che è il codice schermo di M, seguito da 20 (spazio), 30 (0), 34 (4)... ecc.

Modifica di una cella di memoria

Come modificare il contenuto di una o più celle di memoria?
È molto semplice: il principio è quello dell'editor di schermo. Supponiamo di voler modificare il contenuto di 0400: portiamo il cursore sulla linea voluta:

[.] : 0400 0D 20 30 ...

Poi sulla colonna voluta

. : 0400 [0]D 20 30 ...

Lì premiamo il valore che vogliamo in 0400, supponiamo che sia 01:

. : 0400 01 [ ] 20 30...

Se vogliamo modificare altri valori lo facciamo, infine 'Return'.
Supporremo di voler mettere 02 in 0401, 03 in 0402 e 04 in 0403:

. : 0400 01 02 03 ...

Dopo il 'Return':

. : 0400 01 02 03 ...

[·] : 0408 20 20 20 ...

ma ABCD è apparso sulla prima riga dello schermo: è la prova che la scrittura ha avuto luogo, poiché 01, 02, 03, 04 sono i codici schermo di A, B, C, D.

Domanda: cerco di fare la stessa cosa in 0410, cioè metto 01, 02, 03, 04 in 0410. Dovrebbe apparire un secondo ABCD sulla prima linea dello schermo; o non c'è niente.
In effetti, si può sperare che le scritture siano state fatte in memoria e i caratteri ABCD dovrebbero apparire. Infatti, è la memoria di schermo che ci prende in giro: i caratteri sono scritti in blu su blu! Abbiamo scritto solo il codice carattere nella memoria di schermo. Avremmo dovuto scrivere anche il loro codice nella memoria di colore che è in $D800.

Fate: .M D800 - D820

Ottenete, in D810, D811 ecc. valori che finiscono col 6 (conta solo la seconda cifra esa). Sostituite i 6 con delle E. Le vostre lettere dovrebbero apparire. Potete anche avere le lettere di colore diverso se mettete dei valori differenti in D810, 11, 12, e 13.
Ma allora perché il primo ABCD è apparso?
Perché andava a sostituire una scrittura già presente sullo schermo, quindi la memoria colore aveva già dei valori utili.

Caricamento di un programma

Ora stiamo per mettere in pratica ciò che precede inserendo un minuscolo programma in memoria.
Questo programma è estremamente semplice: fa la somma del contenuto di due celle di memoria N1 ed N2 e mette il risultato in una terza, R.
Scriveremo dapprima il programma in assembler simbolico. Cominciamo con qualche direttiva, ovviamente per specificare dove sarà installato il programma.
Scegliamo qui di iniziare in C000 (49152), che è una zona tranquilla.

```
*          =C000   ; fissa l'origine
N1         *=*+1   ; riserva spazio
N2         *=*+1   ; per le
R          *=*+1   ; variabili
INIZIO     LDA N1  ; N1 in A
           CLC     ; riporto =0
           ADC N2  ; somma N2 in A
           STA R   ; memorizza in R
           BRK     ; fine programma
```

Questo programma dovrebbe spiegarsi da solo. Gli unici due punti da specificare sono **CLC** che è dovuto al fatto che **ADC** fra la somma comprendendo il riporto precedente e **BRK**: se si vuole che il monitor riprenda il controllo dopo l'esecuzione di un programma, quest'ultimo deve terminare con **BRK**.
Dobbiamo ora fare l'assemblaggio a mano del nostro programma. Per questo, la tabella dell'appendice 2 è indispensabile.
* = C000, l'assemblaggio inizierà all'indirizzo C000. Ne prendiamo nota.
N1 *=*+1 riserva un byte per N1. N1 sarà in C000, lo annotiamo, e siamo pronti ad assemblare in C001.
Le due direttive seguenti installano N2 ed R in C001 e C002.
Allora INIZIO si assembla a partire da C003:

```
C000  N1          * = * + 1
C001  N2          * = * + 1
C002  R           * = * + 1
C003  INIZIO      ...
```

Per assemblare LDA, si reperisce la linea LDA nella tabella dell'appendice II. Sì, ma quale colonna? dipende dal modo di indirizzamento. Qui, l'indirizzo cercato è C000, che non può essere di pagina zero. L'indirizzamento è diretto, da cui il codice operativo AD, l'istruzione completa AD 00C0 e il numero di Byte utilizzati: 3. Si deduce che la prossima cella libera per l'istruzione che segue è: C003 + 3 = C006.
L'assemblaggio prosegue secondo gli stessi principi:

```
C003  AD 00 C0  INIZIO   LDA N1
C006  18                 CLC
C007  6D 01 C0           ADC N2
C00A  8D 02 C0           STA R
C00D  00                 BRK
```

Per caricare il programma, non ci resta che decidere il valore da mettere in C000 (per N1), C001 (per N2). Per N1 e N2 prendiamo 15 e 28 perciò, una volta caricato il programma, avremo:

```
.:C000 15 28 00 AD 00 C0 18 6D
.:C008 01 C0 8D 02 C0 00 00 00
```

Domanda: volete ottenere il listing, qui sopra, sulla stampante?
Premete X per uscire dal monitor. Premete in modo diretto del Basic

OPEN 4,4 : CMD 4 : SYS 8*4096

ed infine M.

Comando G

Si scrive nella forma: **. G ␣ indirizzo**
Ha l'effetto di lanciare l'esecuzione a partire dall'indirizzo dato.
Così, per eseguire il nostro programma qui sopra, si farà:

.G C003 'Return'

Viene visualizzato:

```
B*
   PC  SR AC XR YR SP
.;C00D 30 3D E6 00 F6
```

in cui l'elemento interessante è che c'è 3D (15 + 28 in esa) nell'accumulatore. Si verifica che tale valore è anche in R (C002) con l'aiuto del comando M.
Si vede come l'esecuzione di un programma può essere seguita inserendo delle istruzioni **BRK** che restituiscono il controllo al monitor, ciò che permette di esaminare il contenuto dei registri o delle celle di memoria desiderate.
Esiste una forma abbreviata del comando: **G** da solo che fa partire l'esecuzione dall'indirizzo contenuto attualmente nel PC.

Comando R

Permette di visualizzare il contenuto dei registri del 6502 (stessa visualizzazione di quando si torna al monitor dopo un'istruzione BRK).
Il contenuto dei registri può essere modificato. Bisogna procedere, come per la memoria, con l'editor di schermo con l'aiuto del cursore.
Non dimenticate che è 'Return' che convalida la linea visualizzata: quindi la modifica non diventa effettiva che dopo aver fatto 'Return'.

Al momento del prossimo comando G, i valori visualizzati, eventualmente modificati, saranno nei registri propriamente detti e il programma partirà con tali valori.

Esempio

Caricate il programma:

```
                    * =   $C050
C050  8E 55 C0      STX $C050
C053  00            BRK
```

Modificate X da come è mostrato, con un comando R, in modo che valga BB.
Eseguite il programma (G C050). Verificate che ci sia BB in C050 (M C055 - C056).

Comando X

Questo comando fa uscire dal monitor per ritornare al Basic: si ottiene la visualizzazione di READY.

Comando S

Permette di salvare un programma su cassetta o disco, come se si trattasse di un programma Basic.
La differenza è che bisogna precisare gli indirizzi di inizio e di fine programma (per un programma Basic, il sistema li conosce automaticamente).
Il comando si scrive:

.S "nome", **perif, ind1, ind2**

- nome è il nome che si vuol dare al programma. Per il disco è numero disco:nome;
- perif è il numero della periferica 01 o 02 per la cassetta, 08 per il disco (lo 0 è obbligatorio: occorrono due cifre);
- ind1 è l'indirizzo d'inizio;
- ind2 è l'indirizzo di fine +1. I due indirizzi devono essere di quattro cifre.

Esempio

- per la cassetta scriveremo:
  .S "SOMMA",01,C000,C00E
- per il disco:
  .S "0:SOMMA",08,C000,C00E

Esercizio 2.11 *Perché C00E?*

Comando L

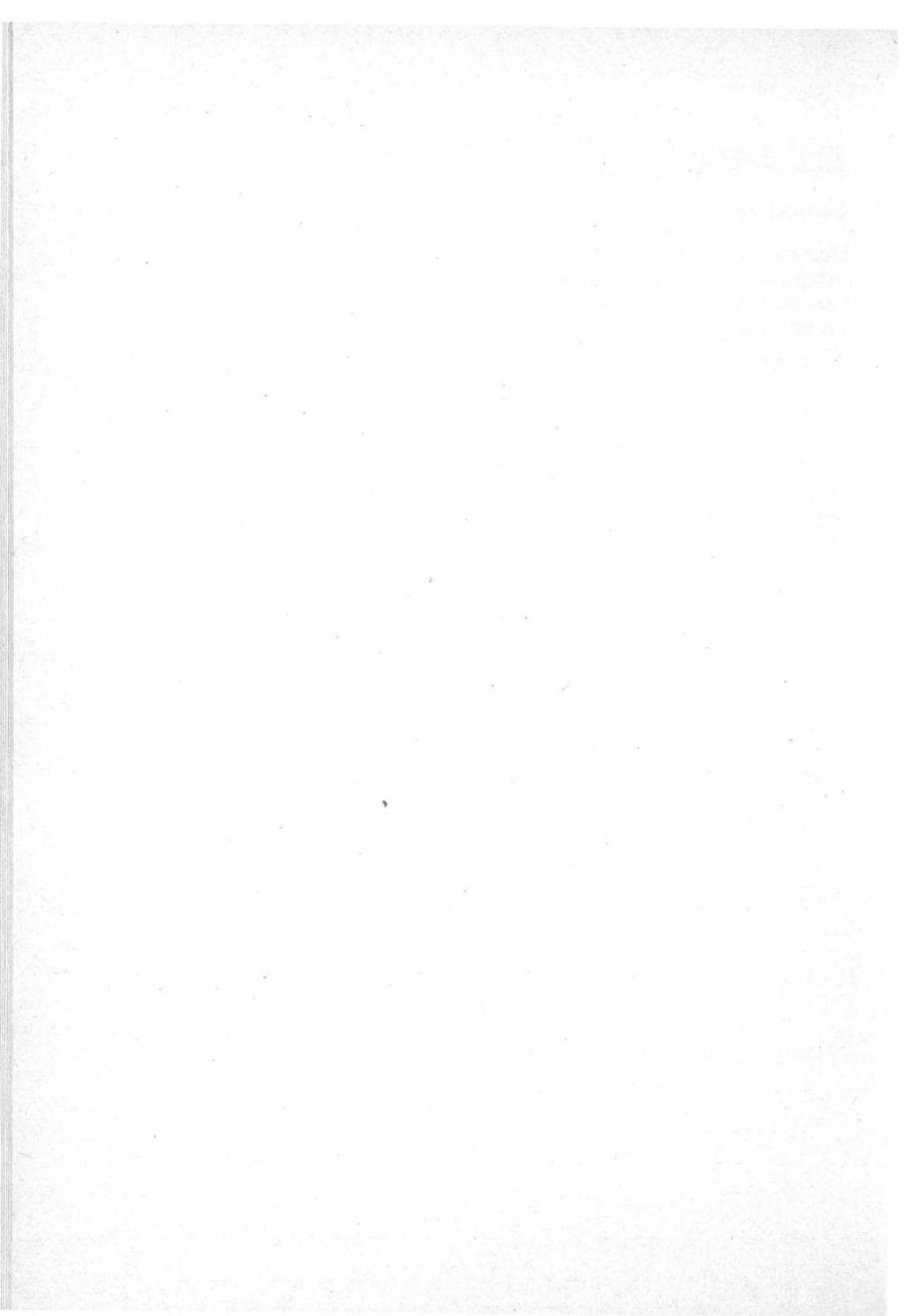
Questo comando è il duale del precedente. Permette di caricare un programma nella zona di memoria che si era precedentemente specificato durante il salvataggio.
Il comando si scrive:

.L "nome",perif

Esempio

.L "SOMMA",01 cassetta
.L "0:SOMMA",08 disco

Ora siamo pronti ad abbordare seriamente la programmazione in assembler poiché disponiamo delle utilities necessarie che formano il monitor.

Capitolo 3

Programmi elementari e complicati

Siamo ora ai piedi del muro: si tratta di passare alla programmazione effettiva.
Abbiamo le utilities che ci servono per questo: fintanto che i nostri programmi non siano troppo lunghi, possiamo farne l'assemblaggio a mano e caricarli in memoria con l'aiuto del monitor.
Per i programmi più lunghi, utilizzeremo l'assembler descritto al capitolo seguente.
Per ora, cominciamo coi programmi aritmetici più elementari, poi esploreremo le utilizzazioni pratiche dei modi di indirizzamento del 6502: sappiamo che il 6502 è particolarmente ricco da questo punto di vista.

PROGRAMMI ARITMETICI ELEMENTARI

Somma su 8 bit

Il programma aritmetico più semplice l'abbiamo già visto al capitolo 2, a titolo d'esempio dell'utilizzazione del monitor è la semplice addizione di due numeri.
Tuttavia si manifesta una particolarità importante del 6502: lo stato precedente del flag di riporto (C) interviene nel risultato. L'effetto esatto dell'operazione ADC è $A \leftarrow A+M+C$ (A = accumulatore, M = memoria relativa). Se si vuole un'addizione pura, è sufficiente annullare C (Carry = riporto) con un'istruzione CLC precedente. È quello che abbiamo fatto.
Sì, ma perché il 6502 non ha un'istruzione di addizione semplice? È a scopo di economia: non dimentichiamo che il numero di combinazioni

istruzione-modo d'indirizzamento è limitato a 256 poiché il codice operativo è di un byte.
Gli inventori del 6502 hanno scelto di "impoverire" il set di istruzioni per arricchire i modi di indirizzamento, che è un punto di forza di questo microprocessore.
Peraltro è indispensabile disporre di un'istruzione che faccia intervenire il riporto per poter eseguire delle somme in doppia precisione, come ne vedremo più avanti.
Un parametro importante che caratterizza una istruzione è la sua azione sui flag di stato. Si vede, dalla tabella dell'appendice 2, che ADC agisce sui flag:

N (messo a 1 se il risultato è negativo);
Z (messo a 1 se il risultato è nullo);
C (messo a 1 se si è generato un riporto);
V (messo a 1 se si è prodotto un superamento di capacità "overflow").

Per allenarci a interpretare queste indicazioni, eseguiremo il programma di pagina 39 per differenti coppie di valori di N1 ed N2 (indirizzo C000 e C001).
Nell'accumulatore (sotto AC), si vede che il numero visualizzato è la somma dei nostri due numeri.
Per interpretare il contenuto del registro di stato (sotto SR), ricordiamo il suo schema:

| NVXBDIZC | (X è inutilizzato e vale sempre 1).

Per la prima coppia di valori (15 e 28), si ottiene: AC = 3D (è proprio 15+28 esa) e SR=30 esa = 0011 0000, cioè:

N = 0 (il risultato è positivo)
VV = 0 (non c'è overflow)
X = 1 (deve sempre essere 1)
B = 1 (si è appena eseguita una BRK)
D = 0 (non si è in modo decimale)
I = 0 (gli interrupt sono abilitati)
Z = 0 (il risultato non è nullo)
C = 0 (non c'è riporto)

Proviamo altri esempi:

FF + 01 = 00 SR = 33

Si è fatto (–1)+1. Si ottiene logicamente 0 e Z=1.
Si vede anche che c'è un riporto (C=1). Infatti, in esa, FF+01 fa 100.
C'è dunque un riporto, ignorato nel risultato, come è regola nel complemento a due.

54 + 36 = 8A SR = F0

Si trova Z=0 e C=0: risultato non nullo e niente riporto. Ma abbiamo anche N=1 (risultato negativo: 8A=–138 decimale) e V=1. C'è, in effetti, overflow: 54+36>7F(127) ecco il perché una somma di due numeri positivi appare negativa; è, per definizione, una condizione di overflow.

FF + FE = FD (–1)+(–2)=(–3) SR = B1 = 1011 0001
C = 1 : riporto ignorato
Z = 0 : risultato non nullo
V = 0 : non c'è overflow
N = A : risultato negativo (–3)

Esercizio 3.1 *Fare la somma C0+A0. Prevedere il risultato e il registro di stato che deve apparire. Interpretarlo. Verificare poi sulla macchina.*

Alcuni modi d'indirizzamento

Conosciamo ora 5 istruzioni: **LDA, STA, CLC, ADC e BRK**.
CLC e BRK hanno, per solo modo di indirizzamento, il modo implicito: infatti, rappresentano un'operazione puramente interna che non ha bisogno di precisare un indirizzo; stanno quindi in un byte.
CLC (forza C a 0) appartiene al gruppo delle istruzioni esplicite incondizionate sui flag di stato.
A questo stesso gruppo appartengono **SEC** (forza C a 1), **SED** e **CLD** (forza a 1 o a 0 il flag D), **SEI** e **CLI** (forza a 0 o a 1 il flag I) e **CLV** (forza a 0 il flag V; **SEV** non esiste).
Ricordiamo che altre istruzioni sono in grado di agire sui flag, l'abbiamo visto per **ADC**. Ma, in tal caso, i flag sono settati non incondizionatamente, ma secondo le operazioni e i dati relativi.

Domanda: nell'esercizio di allenamento precedente, esaminiamo il registro di stato per vedere come agisce **ADC.** Ma lo stato che osserviamo è ottenuto dopo due altre istruzioni **STA** e **BRK**.

- **STA** non agisce su alcun flag (cfr. appendice 2).
- Quanto a **BRK** salva il registro SR nella pila ed è lo stato così come è stato salvato che il monitor ci permette di esaminare. Si può dunque dire che è lo stato risultante dopo **ADC** che noi osserviamo. Occorre, naturalmente, stare attenti a non intercalare delle istruzioni che modifichino SR se vogliamo osservare l'azione di un'istruzione sui flag.

Le altre tre istruzioni sono state utilizzate nel modo di indirizzamento più naturale, l'indirizzamento diretto in cui il codice operativo è semplicemente seguito dall'indirizzo in due byte. Ricordatevi l'inversione byte basso-byte alto dei due byte dell'indirizzo. Queste istruzioni sono suscettibili di altri modi di indirizzamento: ora ne vedremo qualcuno.

Indirizzamento immediato

Per manipolare un numero conosciuto in quanto tale, si dice che si tratta di una costante non è l'indirizzo che figura nell'istruzione, ma il numero stesso.
Questo si chiama modo di indirizzamento immediato. È segnalato in assembler simbolico dal segno #.

Esempio

Sommiamo 38 e 24 (esa) nell'accumulatore; non memorizziamo il risultato perché il monitor ci permette di osservarlo nell'accumulatore. Il programma comincia in C000:

```
              * =  $C0000
C000  A9  38  LDA #$38
C002  18      CLC
C003  69  24  ADC #$24
C005  00      BRK
```

LDA e **ADC** si assemblano questa volta in due byte. Come sempre, i codici operativi ci sono forniti dall'appendice 2. Otteniamo come risultato: AC = 5C e SR = 30.

Esercizio di allenamento 3.2 *Rifate, questa volta in modo immediato, le addizioni provate precedentemente.*

Indirizzamento pagina zero

Questo modo di indirizzamento permette di guadagnare spazio in memoria e tempo di esecuzione.
È derivato dall'indirizzamento diretto e consiste nel particolarizzare una zona di 256 byte di memoria, la pagina zero.
La pagina zero è l'insieme dei 256 primi byte della memoria, di indirizzamento tra 0000 e 00FF: il byte alto dell'indirizzo è sempre 00 da cui il nome della pagina.
Si può perfettamente indirizzare un byte della pagina zero con l'aiuto dell'indirizzamento diretto: si sarà semplicemente nel caso particolare in cui il terzo byte dell'istruzione è 00.

Esempio

```
AD 01 00   LDA $0001
```

L'indirizzamento pagina zero consiste nell'evitare di scrivere tale 00, a condizione di avvisare la macchina modificando il codice operativo. Così, LDA $0001 si assembla A5 01. Si risparmia un byte e un ciclo macchina durante l'esecuzione.

Esempio

Sommiamo i contenuti delle celle 0011 e 0002; il risultato sarà lasciato in accumulatore.

```
              * =  $C000
C000  A5  11  LDA  $11
C002  18      CLC
C003  65  02  ADC  $02
C005  00      BRK
```

Per eseguire questo programma, occorre caricare dei valori nelle celle di memoria 0011 e 0002 (comando M del monitor).
In ragione del risparmio che comporta, l'indirizzamento pagina zero è molto interessante. Ma è limitato poiché la pagina zero contiene solo 256 byte. E perciò bisogna ripartire con molta cura i dati in memoria e mettere in pagina zero solo le variabili più utilizzate se non si possono mettere tutte.
Questo fatto è molto utilizzato dal sistema operativo, quello del C64 naturalmente.
Non abbiamo esaurito, tutt'altro, i modi di indirizzamento del 6502. Ma, prima, terminiamo le istruzioni aritmetiche.

Addizione in doppia precisione

Ecco ora la giustificazione del fatto che l'addizione fa intervenire il riporto.
Supponiamo di voler trattare dei numeri più grandi di 256 (è proprio il caso!). Bisogna allora che i numeri stiano su più byte, diciamo due.

Esercizio di revisione 3.3 *Qual è il campo di variabilità dei numeri, se sono di due byte in complemento a due?*

Il programma di addizione di due numeri di due byte ciascuno è molto semplice: è sufficiente procedere come avete fatto per sommare 36 e 42! Sommate dapprima le due cifre di destra 6+2=8, poi quelle di sinitra con l'eventuale riporto: 3+4=7, da cui il risultato 78.
Ebbene qui si sommeranno prima i due byte di destra (senza riporto, quindi si farà CLC); si memorizzerà il risultato nella parte destra dei due byte relativi al risultato.
In seguito, si sommano i due byte di sinistra col riporto.
Facciamolo sull'esempio 1936+2542. Prenderemo gli operandi in modo immediato. La parte sinistra del risultato (che è calcolata per ultima) resterà nell'accumulatore AC. La parte di destra sarà salvata nel registro X grazie all'istruzione **TAX**, che ci permette di scoprire una nuova categoria di istruzioni: le istruzioni di trasferimento tra registri.

Queste istruzioni sono molto semplici: il loro codice è del tipo Tab che si legge "trasferisci il contenuto di a in b" e si assemblano in un solo byte (indirizzamento implicito). Controllate nell'appendice 2, quali trasferimenti esistono.
Da cui il programma:

```
C000 A9 36 LDA #$36
C002 18    CLC
C003 69 42 ADC #$42
C005 AA    TAX
C006 A9 19 LDA #$19
C008 69 25 ADC #$25
C00A 00    BRK
```

Si vede che si ottiene X=78 e AC=3E, da cui il risultato 3E 78.

Esercizio 3.4 *Fate la somma dei numeri esadecimali 35428A e 274BCD. Il risultato sarà contenuto in AC X e Y (da sinistra a destra).*

Esercizio 3.5 *Sul 6502 non esiste l'istruzione TXY. Come si può costruirla?*

Domanda: si potrebbe invertire la posizione di **CLC** e del primo **LDA**?

– Sì perché LDA non modifica il flag C.

Notate che c'è una sola **CLC** per la somma dei byte meno significativi: per le somme seguenti, bisogna al contrario mantenere il riporto precedente perché interviene nell'addizione.

Sottrazione

L'istruzione di sottrazione del 6502 è SBC. Anch'essa fa intervenire il riporto, ma attenzione, esso obbedisce allo schema: $A \leftarrow A - M - \overline{C}$. È l'opposto $\overline{C}$ del riporto che interviene.
Di conseguenza, se si vuole che il riporto non interventa (esempio: per la prima di una serie di sottrazione in doppia precisione), è SEC che bisogna fare (SEC mette a 1 C).
Se, al termine di una sottrazione, il flag C è a 0, significa, non che non c'è riporto, ma, al contrario, che ci sarà un prestito al passo successivo (se si è in doppia precisione). Se C=1, significa che non c'è prestito, quindi che il numero sottratto dall'accumulatore era più piccolo di quest'ultimo. Verifichiamolo su qualche esempio.
Come per l'addizione, sottraiamo il numero contenuto in **N2** da quello contenuto in **N1**. Esaminiamo il risultato nell'accumulatore, così come il registro di stato.

Il programma da caricare è:

```
                        N1    = $C000
                        N2    = $C001
                        *     = $C002
C002  AD 00 C0              LDA N1
C005  38                    SEC          ; QUI CI VUOLE SEC
C006  ED 01 C0              SBC N2
C009  00                    BRK
```

Proviamo con N1=30 e N2 =40, otteniamo AC=F0 (che rappresenta giustamente –16) e SR=B0. Questa volta, N=1 e C=0 quindi $\bar{C}$=1: c'è prestito.

Per N1 = FF0 ; N2 = E0 otteniamo AC = 10 e SR = 31
Per N1 = E0 ; N2 = F0 otteniamo AC = F0 e SR = B0
Per N1 = 90 ; N2 = E0 otteniamo AC = B0 e SR = B0
Per N1 = 90 ; N2 = 20 otteniamo AC = 70 e SR = 71

L'esempio merita un ripensamento: in effetti, troviamo N=0, quindi un risultato positivo, ed è stupefacente poiché abbiamo fatto un numero negativo meno qualche cosa. È un caso tipico di overflow e, infatti, troviamo V=1. Il risultato AC=70 è corretto se si considera il numero senza segno (da 0 a 255); è sbagliato in complemento a 2, ciò è segnalato dal flag V=1. Infine abbiamo C=1 che significa N1>N2 (numeri considerati senza segno).

Esercizio 3.6 *Interpretare la sottrazione N1(=0C0;–N2(=40) e C0–41.*

Note sul modo di operare le verifiche.

1. Bisogna fare G C002 (e non G C000) poiché il programma effettivo inizia in C002.
2. Il monitor linguaggio macchina procede con l'editor di schermo. Per dare il valore d N1 ed N2, si fa M C000- C001, poi G C002, da cui la visualizzazione:

   .M C000 - C001
   .: C000 ......
   .G C002
   B*
      PC SR...
   .; ......
   ▪ cursore.

Per dare una nuova coppia, basta riportare il cursore sulla linea .: C000... e cambiare i valori poi premere 'Return'. Col cursore su .G C002 'Return' fa eseguire il programma, e così di seguito.

Esercizio 3.7 *Fare la sottrazione in doppia precisione dei numeri contenuti in C000; C001 e C002; C003. Risultato in AC (byte alto) e X (byte basso).*

Questo termina le istruzioni puramente aritmetiche a due operandi del 6502. Sembra quindi che ve ne siano solo due. Ebbene è sbagliato. Ve ne sono infatti 4 grazie al modo decimale.

Il modo decimale

Abbiamo visto che un byte poteva rappresentare un numero binario: è la rappresentazione più naturale con la quale abbiamo fatto i calcoli precedenti.
Ma un byte può anche rappresentare un numero in decimale codificato binario, cioè ogni gruppo di 4 bit rappresenta una cifra decimale da 0 a 9.
Per esempio 19 esa = 0001 1001 rappresenta 19 decimale, mentre, normalmente, 19 esa rappresenta 25.
Se si vogliono fare dei conti coi numeri espressi in decimale codificato binario, si presenta una difficoltà.
Per esempio, se vogliamo fare 19+1, l'unità aritmetica che è "abituata" a calcolare il binario darà 1A, che è normale. Ma in questo caso avrebbe dovuto dare 20.
La maggior parte dei microprocessori hanno una istruzione **DAA** che "adatta" il risultato. Il 6502 procede in modo più comodo. La sua unità aritmetica può svolgere le istruzioni ADC e SBC, in due maniere, secondo il valore del flag D.

Se D=0, è in modo binario, 19+1=1A
Se D=1, è in modo decimale, 19+1=20

L'interesse del modo di procedere del 6502 è che basta forzare a 1 il flag D con un'istruzione **SED** una volta per tutte: tutte le operazioni ADC e SBC e la serie di operazioni in doppia precisione saranno effettuate in modo decimale. Si può dire che il 6502 è il miglior microprocessore per programmare un terminale di un punto di vendita.
I flag aritmetici hanno un comportamento un po' delicato: solo C è veramente significativo. Poiché i numeri sono considerati senza segno (99 > 0), N e V non hanno significato. Z è falso: 99+1 dà 0 e un riporto, ma Z resta a 0.
Ultima precauzione col C64: in un programma che funziona in intera-

zione col Basic non dimenticate di ritornare al modo binario con una CLD prima di tornare al Basic; infatti il sistema operativo del C64 funziona solo in modo binario.

Esercizio 3.8 *Eseguite l'operazione decimale in doppia precisione 1956+2341–1028.*

OPERAZIONI ARITMETICHE UNARIE

Le operazioni viste precedentemente avevano due operandi: l'accumulatore e una cella di memoria; sono delle operazioni binarie.
C'è un'altra categoria di operazioni, le operazioni unarie, che hanno un solo operando. Il loro effetto è di apportare una modifica all'operando, per esempio aggiungergli 1. Gli operandi possibili sono l'accumulatore **A**, i registri **X** o **Y** o una cella di memoria.
Vediamo ora le operazioni di questa categoria.

Incremento-decremento

Queste istruzioni aggiungono (INC, INX, INY) o sottraggono (DEC, DEX, DEY) 1 a una cella di memoria o ad X o Y. Notate che non ci sono operazioni di questo tipo sull'accumulatore, perché al posto di fare "INC A", è possibile fare:

```
CLC         SEC
ADC  #1     ADC  #0
```

Gli incrementi decrementi di X e Y sono soprattutto utili per percorrere un array grazie all'indirizzamento indicizzato.

Scorrimento-rotazione

Queste istruzioni (**LSR** scorrimento a destra, **ASL** scorrimento a sinistra, **ROR** rotazione a destra, **ROL** rotazione a sinistra) agiscono sia sull'accumulatore che su celle di memoria.
Il bit uscente si ritrova nel riporto. Negli scorrimenti del 6502, il bit entrante è sempre 0. Nelle rotazioni, il bit entrante è il vecchio valore del riporto.
Questo può servire per fare uno scorrimento in doppia precisione.

Esempio

Per moltiplicare per 2 il numero su 16 bit che si trova in C000 (basso) e C001 (alto) si fa:

```
C002 0E 00 C0  ASL BASSO
C005 2E 01 C0  ROL ALTO
C008 00        BRK
```

Esercizio 3.9 *Si abbia C001=00abcdef, C000=ghijklmn. Eseguire, step by step (passo a passo), il programma qui sopra.*

Le istruzioni di scorrimento permettono di esaminare, uno ad uno, tutti i bit di un byte. Serve per le moltiplicazioni.
Un altro uso importante concerne la trasmissione di dati in serie in cui si inviano i bit di un dato uno per volta.

Esercizio 3.10 *Si abbia un numero BCD in un byte. Bisogna convertirlo in binario (esempio: 83 deve diventare 53).*

Chiamiamo [u|v] *il numero. Per* [0|v] *la conversione è presto fatta. Per* [u|0] *il problema è che* [u|0] *rappresenta u×16 in binario e u×10 in decimale. Se si nota che* [00u] *(*[u|0] *fatto scorrere a destra due volte) vale u×4, allora u×10 è uguale a 1/2 (*[u|0] *+* [00u]*).*

Se L = [0|v]*, M =* [00u]*, N =* [u|0]*, il numero convertito sarà (M+N) fatto scorrere a destra + L.*
Si supporrà che all'inizio il numero da convertire sia nell'accumulatore. Si metterà L in 0022 e M in 0023 (pensate all'indirizzamento pagina zero); non avete bisogno di indirizzo per M (dove lo mettereste voi?).
Bisogna dire che il tutto si basa sul fatto che un'istruzione di questo tipo agente su una cella di memoria non modifica l'accumulatore.

OPERAZIONI LOGICHE

Queste operazioni sono binarie (A ← A op. M) e sono state viste rapidamente al capitolo 1.
Ricordiamole:

AND effettua l'and logico bit a bit. Può servire per forzare a 0 dei bit isolati.
ORA effettua l'or logico. Può servire per forzare ad 1 dei bit isolati.
EOR effettua l'or esclusivo. Può servire a complementare (a 1) dei bit isolati.

Queste operazioni che permettono di isolare dei bit (si dice che si tratta di una mascheratura) o che permettono di agire su un bit senza toccare gli altri sono utili nell'input-output (ingresso/uscita).

```
0000                  BASSO =   $C000
0000                  ALTO  =   $C001
0000                  *     =   $C002
C002  AD 00 C0              LDA BASSO
C005  49 FF                 EOR #$FF
C007  8D 00 C0              STA BASSO
C00A  AD 01 C0              LDA ALTO
C00D  49 FF                 EOR #$FF
C00F  8D 01 C0              STA ALTO
C012  18                    CLC
C013  69 01                 ADC #1
C015  8D 00 C0              STA BASSO      ; AGGIUNGE 1
C018  AD 01 C0              LDA ALTO
C01B  69 00                 ADC #0
C01D  8D 01 C0              STA ALTO
C020  00                    BRK
```

Esempio

Calcoliamo l'opposto del numero situato agli indirizzi C000 (basso) e C001. Per questo si complementa (a 1) con EOR, poi si aggiunge 0001.

Esercizio 3.11 *Senza utilizzare INC, potreste diminuire i trasferimenti di dati dal programma qui sopra e così ottimizzarlo un po'.*

Esercizio 3.12 *Nel problema dell'esercizio 3-10, bisognava isolare* | u | 0 | *e* | 0 | v | *da* | u | v |. *Degli AND con delle maschere opportune dovrebbero permettere di farlo. Riscrivete il programma. Dite quanti byte e quanti cicli macchina si guadagnano (consultate l'appendice 2).*

Abbiamo visto ora la maggior parte delle istruzioni aritmetiche. Per proseguire, ci occorre poter fare dei test da cui la questione importante delle istruzioni di biforcazione.

SALTI E BIFORCAZIONI

Le istruzioni di salto permettono di simulare delle decisioni nel corso dell'esecuzione del programma.
Come nei linguaggi evoluti, ci sono in linguaggio macchina delle istruzioni di salto incondizionato (l'equivalente di GOTO) e delle istruzioni di salto condizionato.
Le istruzioni di salto incondizionato del 6502 sono:

NOP che significa No OPeration (nessuna operazione), si può interpretare come un salto all'istruzione seguente.

BRK che sarà spiegata più avanti e che, col monitor del C64, s'interpreta come un salto al monitor.
JSR che sarà vista più avanti coi suoi riscontri **RTS** ed **RTI**.
JMP; JMP, LÀ significa saltare all'istruzione la cui label sia LÀ.

Ha due modi di indirizzamento: il diretto e l'indiretto.
In indirizzamento indiretto, si scriverebbe:

JMP (PUNTATORE)

sapendo che in PUNTATORE e in PUNTATORE+1 si trovano rispettivamente il byte basso e il byte alto del vero indirizzo a cui si vuole arrivare.
Le istruzioni di biforcazione sono condizionate. Ciascuna consiste nell'esaminare un flag di stato. Se questo è nella condizione desiderata, si salta all'indirizzo che segue la biforcazione. Insomma, non è molto differente da:

```
IF...GOTO LÀ
...
...
LÀ
```

Il 6502 permette di esaminare i flag N, Z e V, e ognuno per i valori 0 e 1.

BCC	biforcazione	se C = 0	
BCS	biforcazione	se C = 1	
BVC	biforcazione	se V = 0	
BVS	biforcazione	se C = 1	
BPL	biforcazione	se N = 0	quindi risultato positivo o nullo (PLus)
BMI	biforcazione	se N = 1	quindi risultato negativo (MInus)
BNE	biforcazione	se Z = 0	quindi confronto dato Non Eguale
BEQ	biforcazione	se Z = 1	quindi confronto dato Eguale

Grazie a BCC e BCS, si può testare qualsiasi bit di un byte con l'aiuto di scorrimenti.

Esempio

Saltare a **LÀ** se il bit 3 di TRUCCO è a 1:

```
LDA  TRUCCO
LSR  A          ; bit 0
LSR  A          ; bit 1
LSR  A          ; bit 2
LSR  A          ; bit 3 in C
BCS  LÀ
```

È preferibile fare un AND se c'è solo un bit da testare:

LDA #$08 ; Bit 3 a 1
AND TRUCCO
BNE LÀ ; se il bit è a 1 l'operazione ha risultato non nullo

Esercizio 3.13 *Saltare a LÀ se il registro X contiene 0.*

Domanda: come testare un altro flag diverso da N, V, Z o C?

Si può (vedere più avanti) trasferire il registro di stato (P) nell'accumulatore (passando per la pila) e poi testare il bit che si vuole.

Indirizzamento relativo

Le istruzioni di biforcazione hanno assunto un modo di indirizzamento particolare. Non è indicato l'indirizzo assoluto al quale bisogna saltare, poiché ciò renderebbe necessari tre byte per ogni istruzione. Poiché i salti sono molto utilizzati, ciò sarebbe costoso in termini di spazio di memoria. Ma non si può utilizzare l'indirizzamento pagina zero perché è molto raro dover saltare in pagina zero,
Fortunatamente, si è constatato che, spesso, le biforcazioni arrivano a breve distanza dall'istruzione di partenza.
È dunque la distanza relativa al contenuto del PC che è indicata nella forma di spiazzamento con segno che sta su un byte.
Lo spiazzamento è calcolato automaticamente con l'assembler simbolico. Nell'assemblaggio a mano, il calcolo deve essere fatto con molta cura perché gli errori sono fatali: il 6502 salta là dove non dovrebbe e può anche cadere in un posto dove non ci sono istruzioni.
Il punto fondamentale è che l'istruzione è a "tot", cioè il codice operativo è all'indirizzo tot, allora PC vale tot+2 quando la biforcazione viene eseguita, poiché punta già verso l'istruzione successiva (che è in tot+2 poiché la biforcazione occupa due byte).
Se l'istruzione di arrivo è all'indirizzo "centro", allora lo spiazzamento vale:

spiazzamento=arrivo – tot + 2 espresso in complemento a 2.

Esempio

Spiazzamento positivo. Sono in $1000; voglio saltare a $1050.

tot+2=1002, arrivo=1050
da cui spiazzamento 1050−1002=4E

Spiazzamento negativo. Sono in $1000; voglio saltare a 0FE4
da cui spiazzamento=FE4−1002

Calcolo dapprima il valore assoluto dello spiazzamento:

Spiazzamento = 1002 – FEA = 1E
complemento a 1: FF – 1E = E1
complemento a 2: E1 + 1 = E2

Consigliamo vivamente al lettore di seguire sempre questo metodo.

Spiazzamento molto grande
Sono in $1000 e voglio saltare in $1110 da cui lo spiazzamento 1110–1002 = 10E che non sta su un byte.
Allo stesso modo se da $1000 volessi saltare a $10F4, troverei lo spiazzamento 10F4–1002 = F2, che è negativo mentre io so che voglio andare in avanti.
In entrambi i casi, bisogna procedere in modo diverso: una biforcazione semplice è possibile (attenzione il secondo caso è insidioso). Come fare allora? Supponiamo di voler saltare a LÀ se il riporto è a 0, ma che LÀ sia troppo lontano: si sostituisce **BCC LÀ** con:

```
BCC SEGUITO ; prendo la condizione contraria
JMP LÀ      ; nessun problema di distanza
SEGUITO...
```

Alleniamoci ora con qualche uso di biforcazione e di calcoli di spiazzamento corrispondenti.

Incremento di un numero di due byte

Nell'esempio di pagina 67, si è portati a incrementare il numero in doppia precisione. Si comincia con l'incrementare il byte basso. Ma può crearsi un riporto. Però, **INC** non modifica il riporto ma solo Z. Se il risultato è nullo, bisogna incrementare anche il byte alto, altrimenti l'incremento è terminato.
A partire da C012, il programma di pagina 67 si scriverebbe:

```
(C012 8D 01 CO       STA IN ALTO)
C015  EE 00 C0       INC BASSO
C018  D0 03          BNE FINE
C01A  EE 01 C0       INC ALTO
C01D  00       FINE  BRK
```

Lo spiazzamento di BNE è esattamente uguale al numero dei byte da saltare. Se non si fosse voluto introdurre la label FINE si sarebbe potuto scrivere in assembler simbolico: **BNE *+3**. (*)

(*) O *+5 secondo l'assembler utilizzato. Non funziona su EDASM in cui bisogna introdurre la label FINE.

Ciclo d'attesa

Quando una biforcazione è all'indietro, si forma una ciclo.

Attenzione, occorre che il ciclo non duri indefinitamente perché, col monitor del C64, il tasto STOP non funziona.
Alcuni monitor più perfezionati permettono di uscirne, altrimenti bisogna spegnere e riaccendere.
Proviamo un ciclo. Testeremo la cella di memoria SHIFT che vale 1 se stiamo premendo il tasto SHIFT e altrimenti vale 0. Al momento in cui premiamo il tasto il programma termina.
L'indirizzo di SHIFT è $28D sul C64, da cui il programma:

```
                          SHIFT=     $28D
                          *     =    $C000
C000  AD 8D 02            TEST  LDA  SHIFT
C003  F0 FB                     BEQ  TEST
C005  00                        BRK
```

Per calcolare lo spiazzamento:

– spiazzamento = C005 – C000 = 05
  complemento a 1 FF – 05 = FA perciò spiazzamento =FB

Quando si esegue il programma, il cursore scompare. Ritorna solo quando si preme SHIFT.

Esercizio 3.14 *Se SHIFT era in pagina zero, come si sarebbe modificato il programma precedente per tenerne conto? Supponete, per esempio, SHIFT=$98.*

LE ISTRUZIONI DI CONFRONTO

Nell'esercizio 3.13, per testare se X era nullo, abbiamo dovuto prima trasferirlo nell'accumulatore. Altrimenti, il flag Z non avrebbe dato il valore di X che a condizione di aver appena calcolato X stesso. L'inconveniente è che distruggiamo il valore dell'accumulatore. È possibile evitare ciò utilizzando un'istruzione di confronto.
Un'istruzione di confronto fa la sottrazione tra uno dei registri A (CMP), X (CPX), o Y (CPY) e una cella di memoria, e forza i flag di stato di conseguenza (Z=1 se il registro e la memoria sono eguali: la sottrazione dà allora zero). Ma tale sottrazione viene svolta in modo virtuale, cioè il risultato non è conservato: nessun registro viene modificato.

Possiamo quindi scrivere, per l'esercizio 3.13:

```
CPX   #$0  ; confronta X a 0
BEQ   LÀ   ; se X=0, Z varrà 1
```

né A, né X sono mutati.

Questo è molto interessante per fare dei confronti in serie. Per esempio, un carattere è appena stato caricato in accumulatore. Se questo carattere è +, bisogna saltare a 'SOM', –, a DIF, *, a MOLT e /, a DIV. Se non è nessuno dei quattro, a ERR.

```
LDA CAR
CMP #'+'        ;  confronta A al carattere +
BEQ SOM
CMP #'–'        ;  confronta A –
BEQ DIF
CMP #'*'
BEQ MOLT
CMP #'/'
BEQ DIV
; TRATTAMENTO ERRORE
```

Il contenuto dell'accumulatore resta intatto per i confronti seguenti.
Si può solo testare l'uguaglianza? Nient'affatto, benché essa sia la più frequente. Si possono testare anche l'inferiorità e la superiorità, ma il 6502 setta i flag in modo particolare nelle istruzioni di confronto.
Dapprima i numeri confrontati sono considerati come senza segno, da 00 a FF. In seguito, 40 per esempio, sarà considerato come inferiore a C0, mentre se si considerasse il segno 40 che è positivo sarebbe maggiore di C0 che è negativo.
Quindi, il flag V resterà immutato. Quanto a N, non sempre rifletterà il segno del confronto. Avrà un valore corretto solo nel caso in cui il confronto dia lo stesso risultato considerando i numeri sia con segno, che senza segno, quindi, in pratica, quando si confrontano due numeri positivi e che non ci sia overflow.
Gli unici due flag "fedeli" sono Z che vale 1 se i due numeri confrontati sono uguali a C che vale 1 se il registro è maggiore od uguale alla memoria.
Così, CMP M; confronta A ed M, dà luogo ad una tabella del genere:

	N	*V*	*Z*	*C*
A < M	?	–	0	0
A = M	0	–	1	1
A > M	?	–	0	1

Si può, in tutti i casi, prevedere esattamente l'effetto sui flag facendo la somma: registro + complemento di M.
Per esempio, supponiamo che A contenga 0, e M contenga C0. Il complemento di M è:

(FF–CO)+1=3F+1=40

0+40 dà 40 positivo donde N=0, Z=0 e C=0 (niente riporto) C=0 mostra che 0 è minore di C0 considerato senza segno. Qui, N è, accidentalmente, corretto: 0>C0 (negativo) da cui il risultato positivo. Ma se A contenesse 0 e M 80 (–128) cosa succederebbe?

$\bar{M}$=(FF–80)+1=7F+1=80

0+80 dà 80 negativo, da cui N=1, Z=0 e C=0.
N non è più corretto: si trova 0–(–128) negativo! È infatti una situazione di overflow.

Esercizio 3.15 *Scrivete un programma che verifichi l'effetto dei confronti. Utilizzatelo nei differenti casi.*

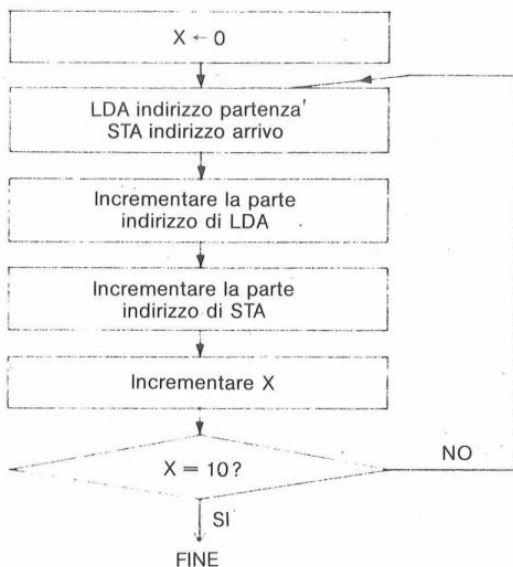
Dopo un confronto, se si vuole una biforcazione verso LÀ se il registro ≥ M, si farà **BCS LÀ**. Se si vuole una biforcazione se il registro è < M, si farà **BCC LÀ**.

Esercizio 3.16 *Dopo un* **CMP**, *saltare a LÀ se A>M.*

Esercizio 3.17 *L'indirizzo TASTO (203 o $CB) contiene 64 se non si sta premendo alcun tasto, altrimenti un codice rappresentativo del tasto. Scrivete un programma che conta (in X) il numero di volte che avete premuto un tasto prima di SHIFT (653 o $28D).*

TRASFERIMENTO DA UNA ZONA DI MEMORIA IN UN'ALTRA

Questo problema si pone spessissimo. Vedremo che è la base per ottenere un disegno animato rapido sullo schermo. Per il momento, ci permetterà di introdurre i modi di indirizzamento indicizzato e indiretto.
In un primo tempo, trasferiremo un gruppo di 10 byte dagli indirizzi C03A a C043 nella zona da 0400 a 040A (tutti indirizzi in esadecimale).
Ora, gli indirizzi da 0400 (a 07E7) corrispondono alla memoria di schermo del C64. Quindi, se in C03A e seguenti, mettiamo 01, 02, 03... 0A che sono i codici schermo di A, B, C... J vedremo apparire in alto

allo schermo, ABCDEFGHIJ, che proverà il buon funzionamento del programma.
Possiamo effettuare un tale trasferimento con quello che sappiamo già? È sufficiente seguire il **flowchart qui di seguito**.
Il registro X serve da contatore di byte trasferiti. Il trasferimento vero e proprio è la serie di LDA STA. Si passa al byte seguente grazie alla serie di incrementi. Per gli indirizzi, bisogna dapprima aumentare il byte basso che è proprio dietro il codice operativo **LDA** o **STA**. Se il risultato è nullo, bisogna aumentare il byte alto che è il byte seguente.

Il trasferimento è finito se X=10 (CPX), da cui il programma:

```
0000                ; TRESFERIMENTO DI 10 BYTE
0000                PART    = $C03A
0000                ARR     = $0400
0000                *       = $C044
C044  A2 00                 LDX #0
C046  AD 3A C0      TR      LDA PART
C049  8D 00 04      TS      STA ARR
C04C  EE 47 C0              INC TR+1        ; ↑ PART BASSO
C04F  D0 03                 BNE IN          ; SALTA INCR
C051  EE 48 C0              INC TR+2          ALTO
```

segue

seguito

```
C054 EE 4A C0   IN    INC TS+1      ; ↑ ARR BASSO
C057 E8               INX
C058 E0 0A            CPX #10       ; X=10?
C05A D0 EA            BNE TR
C05C 00               BRK
```

Notiamo che non abbiamo incrementato che il byte basso dell'indirizzo di arrivo: infatti, tenuto conto della zona di indirizzi di arrivo (0400 - 040A), il byte alto non dovrà mai essere incrementato; è un caso particolare.
Il programma qui sopra si presta a due obiezioni:

1. Poiché prendiamo come contatore un registro di 8 byte potremo spostare solo un blocco di 256 byte al massimo. Questa obiezione sarà tolta più tardi; inoltre in molti casi, 256 byte sono largamente sufficienti.
2. Si tratta di un'obiezione più forte, il programma si modifica da solo durante l'esecuzione. In ogni caso, è una pratica da sconsigliare perentoriamente. Nel caso di un errore che dia un indirizzo scorretto, il programma accederà a una zona di memoria in cui non dovrebbe, ciò può avere delle conseguenze catastrofiche e, in più, è difficilissimo da scoprire quando si cerca di mettere a punto il programma. Per convincervi, tentate di eseguire il programma una seconda volta! D'altra parte, una tale pratica suppone che il programma risieda in RAM per potervi scrivere dentro. È il caso del nostro esempio, ma se noi stiamo scrivendo una routine di sistema che deve risiedere in ROM, non potremmo utilizzare questa tecnica che a rigore è assolutamente da evitare.

Fortunatamente l'indirizzamento indicizzato ci permette di trattare il problema in modo ben più elegante e più sicuro dal punto di vista della chiarezza di programmazione.

Indirizzamento indicizzato-array

Se avessimo modo di mantenere sempre lo stesso indirizzo di partenza e di arrivo, ma potessimo aggiungere il contenuto del registro X a questo indirizzo, cioè se potessimo scrivere:

```
LDA  $C03A  +  X
STA  $0400  +  X
```

il nostro problema sarebbe risolto poiché alla prima iterazione (X=0), il contenuto di C03A sarebbe trasferito in 0400, alla seconda C03B sarebbe trasferito in 0401, ecc.
È precisamente ciò che l'indirizzamento indicizzato permette. Conformemente allo standard della tecnologia MOS, si scrive:

LDA C03A,X o LDA INIZIO,X

e, automaticamente, il microprocessore aggiunge il contenuto del registro X alla parte indirizzo presente nell'istruzione per ottenere l'indirizzo effettivo che sarà inviato sul bus indirizzi:

IE = PI + (X)

Il nostro programma di trasferimento si scrive ora:

```
0000                    ; TRASFERIMENTO DI 10 BYTE
0000                    ; CON INDIRIZZAMENTO INDIRETTO
0000                    PART    = $C03A
0000                    ARR     = $0400
0000                    *       = $C044
C044  A2 00                     LDX #0
C046  BD 3A C0          TR      LDA PART,X
C049  9D 00 04                  STA ARR,X
C04C  E8                        INX
C04D  E0 0A                     CPX #10          ; X=10?
C04F  D0 F5                     BNE TR
C051  00                        BRK
```

Il programma è più corto poiché il meccanismo di indirizzamento indicizzato ci evita l'incremento dell'indirizzo: basta il solo incremento di X.

Gli array

Cosa abbiamo fatto in questo programma? Abbiamo "visitato" successivamente ogni byte di una serie consecutiva di byte in memoria. Una serie di questo tipo si chiama array (vettore, tabella). Avete potuto vedere, in Basic, che l'uso di array è estremamente frequente. Ebbene, con l'indirizzamento indicizzato, LDA M,X è l'esatta traduzione di A = M(X): il registro indice svolge il ruolo di indice dell'array.

Allo stesso modo FOR X = 1 TO N

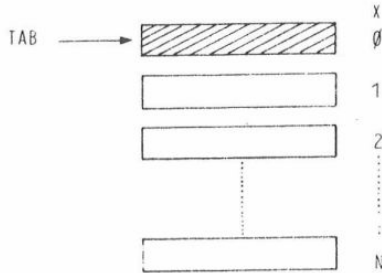
```
        .
        .
        NEXT viene tradotto
CICLO   LDX #1
        .
        .
        INX
        CPX #N+1
        BCC CICLO
```

Esercizio 3.18 *Agli indirizzi 200-258 esa si trova una stringa di 88 caratteri (è il buffer di input del Basic). Scrivete un programma che scriva SI o NO in alto sullo schermo secondo che ci sia o no un segno = nella stringa.*

Ispiratevi al programma precedente per scrivere SI o NO (codice schermo: 13, 09 e 0E, 0F). Il codice ASCII di = è 3D. Per scrivere il programma, eseguite prima un programma in Basic in cui l'unica linea sia 10 INPUT A$ e fornite come risposta una stringa contenente o no un carattere =, poi tornate al monitor (SYS 32768) per fare eseguire il programma di test (inserite il segno = abbastanza avanti nella stringa poiché il monitor altera l'inizio del buffer di input).

L'interesse dell'istruzione CMP appare chiaramente nel ciclo principale di questo programma: si carica una volta per tutte l'accumulatore col carattere = (LDA #$3D) fuori del ciclo ed è conservato per tutta l'esecuzione.
Al contrario, il programma, così com'è scritto, presenta una mancanza grave: contiene due sezioni pressoché identiche: scrivere SI e scrivere NO. L'indirizzamento indicizzato con Y ci fornirà una soluzione.
Infatti, a fianco dell'indirizzamento diretto indicizzato con X, esistono altri tre modi indicizzati: l'indirizzamento indicizzato con Y, l'altro registro indice del 6502 e, per ogni registro indice, un indirizzamento pagina zero indicizzato in cui la parte indirizzo è di un solo byte, il che fa risparmiare tempo e spazio in memoria. Una particolarità dell'indirizzamento indicizzato in pagina zero è che tutto l'array deve essere nella pagina zero perché l'addizione registro+parte indirizzo viene effettuata su di un solo byte.
Questo modo è difficile da usare col C64, essendo che il sistema operativo occupa quasi tutta la pagina zero.
Con l'indirizzamento diretto indicizzato con Y, scriveremo SI o NO con la stessa sezione di programma. La si eseguirà con Y=0 e 1 per scrivere SI e con Y=2 e 3 per scrivere NO, quindi vi si giungerà con Y=0 per SI e Y=2 per NO. Da cui il programma:

```
0000                BUFFER = $200
0000                SCHERM = $400
0000                COLORE = $D800
0000                *      = $C000
C000  13            SINO   .BYTE $13
C001  09                   .BYTE $09
C002  20                   .BYTE $20
C003  0E                   .BYTE $0E
C004  0F                   .BYTE $0F
C005  20                   .BYTE $20
C006  A9 00                LDA #0
C008  A2 27                LDX #39
C00A  9D 00 D8      COL    STA COLORE,X
C00D  CA                   DEX
C00E  10 FA                BPL COL
C010  A2 00         TEST   LDX #0
C012  A0 00                LDY #0       ; PREPARA SI
C014  A9 3D                LDA #$3D     ; SEGNO =
C016  DD 00 02      CICLO  CMP BUFFER,X
C019  F0 07                BEQ SCRIV
C01B  E8                   INX
C01C  E0 59                CPX #89
C01E  90 F6                BCC CICLO
C020  A0 03                LDY #3       ; PREPARA NO
C022  A2 00         SCRIV  LDX #0
C024  B9 00 C0      T1     LDA SINO,Y
C027  9D 00 04             STA SCHERM,X
C02A  C8                   INY
C02B  E8                   INX
C02C  E0 03                CPX #3
C02E  D0 F4                BNE T1
C030  00                   BRK
```

Ecco, 8 byte sono stati guadagnati! La flessibilità procurata dalla presenza di due registri indice ci permette di percorrere due array (SCHERMO e SINO) essendo a diversi livelli nell'uno e nell'altro.
Qui gli spostamenti sono paralleli ma, più in generale, potrebbero essere diversi e anche di senso contrario.
Infatti, finora, abbiamo percorso i nostri array nel senso dell'indice crescente e un CPX (o CPY) ci indicava la fine.
Ma si può percorrere l'array nel senso dell'indice decrescente: basta

inizializzare l'indice con la lunghezza dell'array (= numero elementi) e di fare **DEX** o **DEY** al posto di incrementare.
Ma, a questo punto, se combiniamo bene le nostre cose in modo che l'arrivo dell'indice a 0 segni la fine, l'istruzione **CPX** potrebbe essere evitata. Infatti, se **DEX** dà un risultato nullo il flag Z sarà settato senza che vi sia bisogno di CPX. Basta organizzarsi come nella **figura**: si vede che l'elemento di indirizzo TAB (corrisponde a X=0) non fa, propriamente parlando, parte dell'array che è percorso da:

```
        LDX # N
CICLO     :
          :
        DEX
        BNE CICLO
```

È d'altronde ciò che abbiamo fatto per il ciclo "ausiliario" che fissa il colore della prima linea di schermo, all'inizio del programma.
Il miglioramento con CPX spiega il perché sia più corrente, col 6502, percorrere gli array "in senso inverso". Ma bisogna fare attenzione a non fare un passo di troppo. Vi consigliamo, se vedete le cose meno chiaramente in questo modo, di ritornare alla forma **CPX**.
Supponendo che in 0220 si trovi la lunghezza della stringa nella quale cerchiamo un segno =, il nostro programma si scrive:

```
0000            BUFFER = $200
0000            SCHERM = $400
0000            COLORE = $D800
0000            *      = $C000
C000  13        SINO    .BYTE $13
C001  09                .BYTE $09
C002  20                .BYTE $20
C003  0E                .BYTE $0E
C004  0F                .BYTE $0F
C005  20                .BYTE $20
```

segue

seguito

```
C006  A9 00              LDA #0
C008  A2 27              LDX #39
C00A  9D 00 D8   COL     STA COLORE,X
C00D  CA                 DEX
C00E  10 FA              BPL COL
C010  AE 00 02   TEST    LDX BUFFER
C013  A0 03              LDY #3          ; PREPARA SI
C015  A9 3D              LDA #$3D        ; SEGNO =
C017  DD 00 02   CICLO   CMP BUFFER,X
C01A  F0 05              BEQ SCRIV
C01C  CA                 DEX
C01D  D0 F8              BNE CICLO
C01F  A0 06              LDY #6          ; PREPARA NO
C021  A2 03      SCRIV   LDX #3
C023  B9 FF BF   T1      LDA SINO-1,Y
C026  9D FF 03           STA SCHERM-1,X
C029  88                 DEY
C02A  CA                 DEX
C02B  E0 03              CPX #3
C02D  D0 F4              BNE T1
C02F  00                 BRK
```

Qualche punto merita attenzione in dettaglio, altrimenti i vostri programmi non funzionano e, in linguaggio macchina, il tasto STOP è inutilizzabile per riprendere il controllo.
In $C010, abbiamo ora un LDX in indirizzamento diretto da cui tre byte. In $C023 e C026, abbiamo scritto SINO–1 e SCHERMO–1 a causa dello scorrimento di 1 (Cfr. figura). Abbiamo fatto scorrere il buffer perché il sistema agisce sul primo indirizzo e se avessimo mantenuto 0200, sarebbe stata falsata la lunghezza.

Esercizio 3.19 *Si scriva un testo sulla prima linea dello schermo (40 caratteri). Scrivete un programma che riscriva questo testo invertito sulla seconda linea dello schermo.*

L'indirizzamento indiretto

È tempo ora di togliere l'obiezione secondo la quale il nostro programma di trasferimento non può trattare che degli array con meno di 256 byte. Ci porremo il problema di agire sulla intera memoria di schermo; che è di 1000 byte, dunque più di 256.

Una soluzione possibile è la seguente: tagliare la memoria di schermo in quattro fette di 256 byte. Ogni fetta verrà trasferita da:

```
      LDY #0
CICLO LDA PARTENZA,Y
      STA DESTINAZIONE,Y
      DEY
      BNE CICLO
```

Notate che si trasferisce dapprima il byte n. 0, poi i byte 255, 254... 2, 1. In seguito, bisognerà incrementare l'indirizzo di partenza e l'indirizzo di destinazione (in effetti, i byte alti di questi indirizzi). Ma così si ritorna al programma che si modifica da solo.
Abbiamo visto prima che ci sono delle ottime ragioni per rifiutare una tecnica del genere. L'indirizzamento indiretto ci offre una soluzione corretta.
L'indirizzamento indiretto permette di utilizzare, come indirizzo per un'istruzione, un numero calcolato, cioè la parte indirizzo dell'istruzione e non l'indirizzo dell'operando relativo ma l'indirizzo dell'indirizzo.
Il modo di indirizzamento indiretto puro esiste sul 6502 solo per l'istruzione JMP (salto incondizinato).
Supponiamo che QUI sia l'indirizzo 1000. Allora, JMP QUI (indirizzamento diretto) fa saltare a 1000.
Nell'indirizzamento indiretto scritto JMP (QUI), l'istruzione farà saltare all'indirizzo contenuto in 1000 (e in 1001): se QUI contiene il byte basso di **LÀ**, e **QUI+1** contiene il byte alto, **JMP (QUI)** farà saltare a **LÀ**.
Si vede allora che l'indirizzo a cui si salta può risultare da un calcolo il cui risultato sia stato posto in **QUI** e **QUI+1** prima di eseguire **JMP (QUI)**.
Per le altre istruzioni, l'indirizzamento indiretto offerto dal 6502 è combinato con l'indicizzazione. In entrambi i casi, la parte indirizzo **PI** è in pagina zero.
Nell'indirizzamento indiretto indicizzato con X, l'indicizzazione è effettuata prima dell'indirettezza: in **LDA (PI, X)**, il byte basso dell'indirizzo effettivo è contenuto in **PI+(X)**, il byte alto in **PI+(X)+1**.
Nell'indirizzamento indiretto indicizzato con Y, l'indicizzazione è effettuata dopo l'indirettezza: in **LDA (PI),Y** si trova in **PI** il byte basso e in **PI+1** il byte alto di un indirizzo al quale si aggiunge (Y) per avere l'indirizzo effettivo. Insomma **LDA (PA,X)** accede a un elemento il cui indirizzo è lo Xesimo di un array di indirizzi indiretti, mentre **LDA (PA),Y** accede allo Yesimo elemento di un array il cui indirizzo d'inizio è dato indirettamente da PI (vedi figura 4 capitolo 1).
È quest'ultima forma che ci fornirà la soluzione del nostro trasferimento di più di 256 byte. Vogliamo trasferire la memoria del nostro trasferimento di più di 256 byte. Vogliamo trasferire la memoria di schermo (da 0400 a 07FF) nella zona da C400 a C7FF. Il puntatore di partenza sarà 22,23 (inizialmente 22 conterrà 0, 23 conterrà 80). Il

```
0000                 PART   = $22
0000                 ARR    = $24
0000                 *      = $C000
C000  A2 04                 LDX #4
C002  A0 00          CICLOX LDY #0
C004  B1 22          CICLOY LDA (PART),Y
C006  91 24                 STA (ARR),Y
C008  88                    DEY
C009  D0 F9                 BNE CICLOY
C00B  E6 23                 INC PART+1
C00D  E6 25                 INC ARR+1
C00F  CA                    DEX
C010  D0 F0                 BNE CICLOX
C012  00                    BRK
```

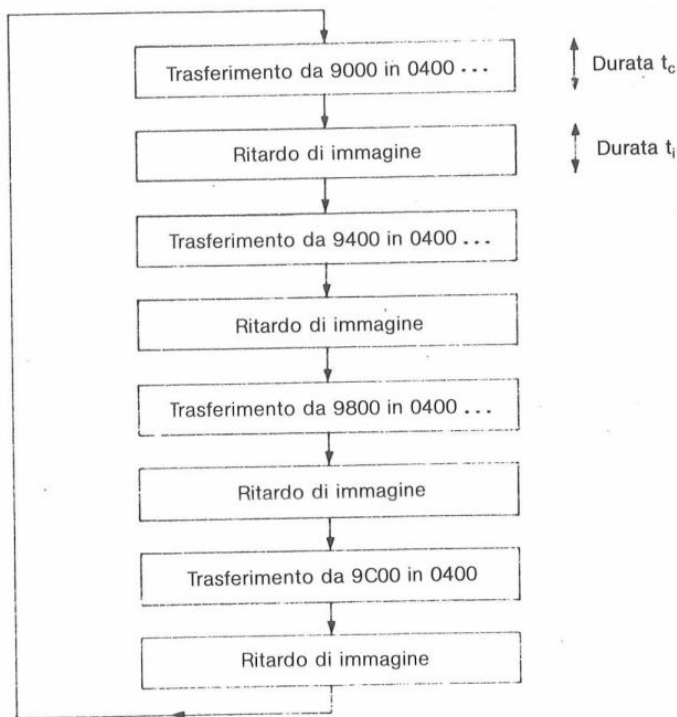
puntatore di destinazione sarà 24,25 (24 conterrà inizialmente 0, e 25 conterrà C4). Da cui il seguente programma.

Per testare questo programma, eseguitelo dapprima con (22,23)=00,04 e (24,25)=00,C4 avendo preparato una schermata facilmente riconoscibile, poi modificate la visualizzazione e eseguite il programma con (22,23)=00,C4 e (24,25)=00,04. La schermata iniziale deve ristabilirsi istantaneamente.

Esercizio 3.20 *Riempite lo schermo istantaneamente con dei caratteri @ (codice 0), o qualsiasi altro carattere che voi preferite.*

DISEGNO ANIMATO

Siamo ora pronti per poter fare un disegno animato abbastanza rapido sullo schermo e ciò ci darà l'occasione di studiare uno dei punti più importanti della programmazione: i sottoprogrammi.
Come fare un disegno animato? Basta decomporre il movimento in un certo numero di tappe e di visualizzare successivamente l'immagine corrispondente ad ogni tappa.
Si pone un problema di tempo: ogni immagine è visualizzata per un tempo t_i; il cambiamento di immagine si fa in un tempo t_c: bisogna che t_c sia piccolo in confronto a t_i e che t_i+t_c (il periodo del fenomeno) non sia troppo grande. Altrimenti, si avrebbe "scintillamento". La visione è accettabile quando si hanno almeno da 15 a 20 immagini al secondo.
Supponiamo di fare un film ciclico (periodico) e di decomporre la scena in quattro immagini diverse.
Il film sarà formato dalla successione delle imagini 1, 2, 3, 4, poi di

nuovo daccapo ecc. Naturalmente il film sarebbe migliore avendo più immagini, ma ciò occupa memoria e noi vogliamo presentare, qui, un programma semplice.
Ogni immagine corrisponde a 1 Kbyte di dati (da 0400 a 07FF). Supponiamo di aver potuto salvare i dati dell'immagine 1 da 9000 a 93FF, i dati dell'immagine 2 da 9400 a 97FF, i dati dell'immagine 3 da 9800 a 9BFF, e quelli dell'immagine 4 da 9C00 a 9FFF.
Il programma di visualizzazione del nostro film sarà allora semplicemente:

Ora sappiamo fare i trasferimenti di zone di memoria. Quindi possiamo fare il nostro disegno animato!

Stima della durata di un programma

In Basic, il trasferimento di 1 Kbyte dura più secondi. Infatti, solo il test del ciclo FOR...NEXT dura 1 ms, allora FOR I=1 TO 1000 : NEXT dura 1 secondo! Aggiungetegli le PEEK e le POKE del trasferimento effettivo e vedrete che non si può, in Basic, fare disegni animati.

Ma il linguaggio macchina ci offre una velocità sufficiente? Per saperlo, bisogna valutare il tempo necessario ad un'esecuzione del programma di pagina 82.
Per questo, abbiamo tutto ciò che ci occorre nell'appendice II che fornisce, per ogni istruzione ed ogni modo di indirizzamento, il numero di cicli macchina occorrenti.
Poiché il 6502 del C64 funziona ad 1 MHz, il numero dei cicli è nel medesimo tempo la durata dei microsecondi. In effetti, il C64 va circa il 10% più lento ma qui non ha imprtanza.
Otteniamo:

```
                            cicli
        LDX =4              2
CICX    LDY =0              2
CICY    LDA (PAR),Y         5
        STA (ARR),Y         6
        DEY                 2
        BNE CICY            3   escluso a fine ciclo in cui è 2
        INC  PAR+1          5
        INC  ARR+1          5
        DEX                 2
        BNE CICX            3   escluso a fine ciclo in cui è 2
        BRK
```

da cui $t_c = 2 + 4 \cdot (2+256 \cdot (5+6+2+3)-1+5+5+2+3)-1$
$= 1+4 \cdot (16+256 \cdot 16) = 16447$
$= 16{,}5$ millisecondi

Ora, se volessimo fare 18 immagini/secondo, ci occorrebbero $t_c+t_i = 1000/18 = 55{,}5$ millisecondi.
Possiamo arrivare ad avere $t_i = 38$ millisecondi.

I sottoprogrammi

Il flowchart che precede è, in effetti, formato da due blocchi eseguiti ciascuno quattro volte: il trasferimento e il ritardo. Li scriveremo quattro volte? Fortunatamente no.
Sapete che in Basic, è possibile utilizzare sottoprogrammi che possono essere chiamati da diversi punti: il sottoprogramma termina con un'istruzione di ritorno che "sa" ogni volta ritornare all'istruzione successiva alla chiamata.
Lo stesso meccanismo esiste nel linguaggio macchina del 6502 (come di ogni processore).

L'istruzione di chiamata si scrive, per esempio, **JSR RITARDO** e l'istruzione di ritorno si scrive **RTS**. Il solo modo di indirizzamento di **JSR** è l'indirizzamento diretto, mentre **RTS** ha un indirizzamento implicito (l'indirizzo di ritorno si trova nella pila).
Il sottoprogramma di trasferimento sarà semplicemente il nostro programma di pagina 82 in cui si cambia unicamente la **BRK** in **RTS**. Prima della chiamata, bisogna preparare gli indirizzi di partenza e di arrivo in 22,23 e 24,25. Si dice che si forniscono "parametri" al sottoprogramma.
Notate, tra l'altro, la comodità che offre l'indirizzamento indiretto per fornire gli argomenti al sottoprogramma.
Per il nostro sottoprogramma di ritardo useremo la tecnica del ritardo programmato. Questa tecnica sarà criticata nel libro sull'input-output. La tecnica è semplice: realizziamo un ciclo nel corso del quale il contenuto di un registro o di una cella di memoria è decrementato da un valore iniziale fino a 0.
Il ritardo ottenuto è proporzionale al valore iniziale. Prendiamo una cella di memoria:

```
RIT DEC  M   6
    BNE  RIT 3
    RTS
```

Se non teniamo conto di RTS, il ciclo dura 9 microsecondi. Il ritardo massimo che può essere ottenuto è di $256 \cdot 9 = 2304$ microsecondi = 2,3 ms.
Dobbiamo dunque decrementare due byte M1 ed M2.

```
RIT DEC  M   1
    BNE  RIT
    DEC  M   2
    BNE  RIT
    RTS
```

M1 è inizializzato a 0 ed è nullo ogni volta che si ritorna a RIT dal ciclo esterno. Il ciclo interno dura dunque 2,3 ms e si inizializza M2 al numero di volte che deve essere percorso cioè ritardo voluto /2,3.
Qui, vogliamo 39 ms da cui M2 = 17 = 11 esa.
Il nostro programma sarà allora:

```
0000             ; DISEGNO ANIMATO
0000                    PART  = $22
0000                    ARR   = $24
0000                    *  = $C000
C000  A2 04  TRASF   LDX #4
```

segue

seguito

```
C002  A0 00        CICLOX LDY #0
C004  B1 22        CICLOY LDA (PART),Y
C006  91 24               STA (ARR),Y
C008  88                  DEY
C009  D0 F9               BNE CICLOY
C00B  E6 23               INC PART+1
C00D  E6 25               INC ARR+1
C00F  CA                  DEX
C010  D0 F0               BNE CICLOX
C012  60                  RTS
C013                      M1  = $3FC
C013                      M2  = $3FD
C013  CE FC 03     RITAR  DEC M1
C016  D0 FB               BNE RITAR
C018  CE FD 03            DEC M2
C01B  D0 F6               BNE RITAR
C01D  60                  RTS
C01E  A9 00        PP     LDA #0          ; PROGRAMMA
                                            PRINCIPALE
C020  85 22                STA PART
C022  85 24                STA ARR
C024  8D FC 03             STA M1
C027  A9 04         DISEG  LDA #$04
C029  85 25                STA ARR+1
C02B  A9 90                LDA #$90        ; IMMAGINE 1
C02D  85 23                STA PART+1
C02F  20 00 C0             JSR TRASF
C032  A9 11                LDA #$11
C034  8D FD 03             STA M2
C037  20 13 C0             JSR RITAR
C03A  A9 04                LDA #$04
C03C  85 25                STA ARR+1
C03E  A9 94                LDA #$94        ; IMMAGINE 2
C040  85 23                STA PART+1
C042  20 00 C0             JSR TRASF
C045  A9 11                LDA #$11
C047  8D FD 03             STA M2
C04A  20 13 C0             JSR RITAR
C04D  A9 04                LDA #$04
C04F  85 25                STA ARR+1
C051  A9 98                LDA #$98        ; IMMAGINE 3
C053  85 23                STA PART+1
```

segue

```
C055  20 00 C0        JSR TRASF
C058  A9 11           LDA #$11
C05A  8D FD 03        STA M2
C05D  20 13 C0        JSR RITAR
C060  A9 04           LDA #$04
C062  85 25           STA ARR+1
C064  A9 9C           LDA #$9C        ; IMMAGINE 4
C066  85 23           STA PART+1
C068  20 00 C0        JSR TRASF
C06B  A9 11           LDA #$11
C06D  8D FD 03        STA M2
C070  20 13 C0        JSR RITAR
C073  4C 27 C0        JMP DISEG       ; RICOMINCIA
```

Non assembliamo questo programma perché ci sono dei miglioramenti evidenti da apportargli. È lo scopo dell'esercizio seguente.

Esercizio 3.21 *Ci sono due miglioramenti evidenti da fare al programma di disegno animato. Fateli.*

Ora che abbiamo il nostro programma di disegno animato, bisogna verificare che funzioni. Tutta la difficoltà sta nel fabbricare le quattro immagini che costituiscono il nostro disegno animato.
La prima cosa da fare è di mettere il programma di disegno animato (pagg. 188, 189) in memoria. Vi raccomandiamo, inoltre, di salvarlo su cassetta o disco, altrimenti rischiate di doverlo riscrivere nel corso della messa a punto!
In seguito bisogna preparare le immagini. Ma poiché abbiamo il sottoprogramma TRASF in memoria, possiamo preparare l'immagine sullo schermo, poi inviarla nella zona di memoria voluta.
Per questo, scriviamo un piccolissimo programma di trasferimento:

```
C050 A9 00     LDA #0
C052 85 22     STA PART
C054 85 24     STA ARR
C056 A9 04     LDA #$04
C058 85 23     STA PART+1
C05A A5 02     LDA #2
C05C 85 25     STA ARR+1
C05E 20 04 C0 JSR TRASF
C061 60        RTS
```

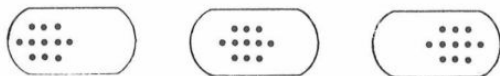
e per trasferire lo schermo in 9000, basta fare POKE 2,144: SYS 49232 (49232=C050 esa).
Non vi resta ora che far partire la vostra immagine. Naturalmente, se avete abbastanza coraggio per fare più di quattro immagini, tutto andrà meglio.

N.B. Non dimenticate di riempire la memoria colore affinché il vostro disegno animato sia visibile. Fate, per esempio, un FOR I= 55296 TO 56295 : POKE I,1 : NEXT.
Un'altra possibilità – che risparmierebbe memoria – sarebbe di dedurre l'immagine n dall'immagine n–1 per mezzo di calcoli effettuati durante la visualizzazione dell'immagine n–1. Questo calcolo sostituirebbe il ritardo. Si può, per esempio, fare appello ad operazioni matriciali per effettuare rotazioni o spostamenti.

Esercizio 3.22 *Quali sono i cambiamenti da fare per avere 6 immagini.*

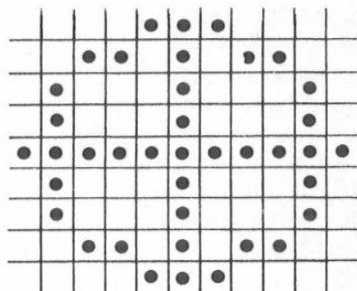
Suggerimenti d'immagini

(È consigliabile lasciare la parte bassa dello schermo libero per poter fare POKE 2,xx:SYS 49232).

Palla che attraversa lo schermo

Palla che gira sullo schermo

(il puntino rappresenta più caratteri come mostrati di seguito).

Ruota che gira

Dettaglio della ruota:

Potete mettere quattro ruote analoghe sullo schermo, rotolanti in senso inverso, o fare le gambe di un giocoliere che fa girare la ruota!

Barra rotante

Potete utilizzare l'alta risoluzione, ma le immagini saranno più lunghe da preparare e ci vorranno 8 K per immagine da trovare in memoria e da trasferire.
Ultimo consiglio fondamentale: potete – e dovete – salvare le vostre immagini su cassetta o disco col comando .S del monitor.

Nota: può essere noioso che il disegno animato (periodico) giri indefinitamente. Potete tescare il tasto SHIFT (cfr. pag. 184) per terminare il programma.

Infine, gli sprite permettono di fare disegni animati in Basic, ma non rotazioni come le facciamo noi.

Esercizio 3.23 *Su quale parametro bisogna agire – se si è preso come esempio la barra rotante – per cambiare la velocità della rotazione? Come cambiare il senso di rotazione?*

MANIPOLAZIONI DELLA PILA

Per mezzo di quale meccanismo l'istruzione **RTS** ritrova l'indirizzo di ritorno dal sottoprogramma?
È semplice: al momento di una **JSR**, il valore di PC è memorizzato in

memoria e viene ritrovato da RTS. Il valore di PC è esattamente l'indirizzo da cui si è fatta la chiamata.
Supponiamo di prendere una zona di memoria unica per questo (occorrono 2 byte) – diciamo gli indirizzi 100 e 101.
Supponiamo che in 1000, si chiami un sottoprogramma (da 2000 a 3000) e che quest'ultimo chiami a sua volta un sotto programma: la cronologia degli eventi è riassunta dalla tabella qui sotto.
Questa struttura non permette, ad un sottoprogramma di chiamarne un altro. È chiaro che occorra più memoria per conservare gli indirizzi di ritorno.

Evento n°		100 101	PC	
1	Chiamata 1° SP...	? / 1000	1000 / 2000	
2	Chiamata 2° SP...	1000 / 2500	2500 / 3000	
3	Ritorno dal 2° SP	2500 / 2500	3500 / 2500	
4	Ritorno dal 1° SP	2500 / 2500	2800 / 2500	Sbagliato

La soluzione consiste nell'utilizzare una pila, cioè una zona di memoria gestita LIFO (Last In – First Out = ultimo entrato – primo uscito).
La pila si comporta esattamente come la pila di documenti che si accumulano sulla scrivania del capo: il primo documento letto è l'ultimo che è stato depositato: la si chiama sommità della pila.
Solo due operazioni sono possibili: posare un documento (questo diventa allora la sommità della pila) o leggere un documento (ciò permette al documento sottostante di diventare la sommità della pila).
Sul 6502, la pila è costituita in memoria fra gli indirizzi esa 100 e 1FF.
Il 6502 possiede un registro interno di 8 bit S, il puntatore alla pila, inizializzato a FF. Ogni volta che si mette in pila S viene decrementato. Ad ogni istante (S) punta al primo posto libero della pila (S)–1 punta alla sommità della pila. Quando si effettua un'operazione sulla pila il 6502 aggiunge automaticamente 100 esa al contenuto del puntatore della pila per generare l'indirizzo effettivo.
Con tale meccanismo, le chiamate di sottoprogrammi progettati qui sopra porterebbero ad una tabella del genere:

Evento	PC	S	Pila:	1FF	1FE	1FD	1FC	1FB	1FA
Prima 1° chiamata	1000	FF		☐					
Dopo 1° chiamata	2000	FD		00	10	☐			
Prima 2° chiamata	2500	FD		00	10	☐			
Dopo 2° chiamata	3000	FB		00	10	(1) 00	25	☐	
Prima del ritorno dal 2° SP	3500	FB		00	10	00	25	☐	
Dopo il ritorno dal 2° SP	2500	FD		00	10	00	(2) 25		
Prima del ritorno dal 1° SP	2800	FD		00	10	00	25		
Dopo il ritorno dal 1° SP	1000	FF		00	10	00	25		

(1) In effetti se il codice operativo **JSR** è in 2500, è 2502 che è messo in pila e dopo il ritorno si ha 2503 perché PC è stato incrementato. Ciò che bisogna ricordare è che il meccanismo assicura il ritorno esattamente dove occorre per eseguire l'istruzione successiva a **JSR**.
(2) 2500 resta nella pila, ma non è più accessibile poiché S è stato aggiornato: è come se fosse stato cancellato.

Si vede dunque che questo meccanismo assicura un funzionamento corretto nelle chiamate nidificate. Quanti livelli si possono avere, cioè quante chiamate si potranno annidare? È semplice, poiché il puntatore alla pila è di 8 bit, la pila contiene al massimo 256 byte che permettono 128 livelli. È più di quanto ci occorra.

Altri usi della pila

Vedremo che la pila serve anche per memorizzare **PC** (ed anche **P**) quando si verificano delle interruzioni.
Di fatto, le interruzioni sono, per molti versi, una sorta di sottoprogramma.
Ma la pila può anche servire da zona di memorizzazione temporanea per delle informazioni da recuperare nell'ordine inverso. Per questo si dispone delle operazioni:

PHA (mette A nella pila)
PLA (toglie A dalla pila)
PHP (mette P nella pila)
PLP (toglie P dalla pila)

Si è portati quando si scrive un sottoprogramma che utilizza i registri, a salvarli all'inizio ed a ripristinarli alla fine, allo scopo di assicurare una certa indipendenza ai sottoprogrammi (si dice modularità). Questo salvataggio si chiama salvataggio di contesto.
Dato che si dispone solo delle quattro istruzioni qui sopra per salvare X e Y, bisogna utilizzare **TXA, TYA** e **TAX, TAY**.

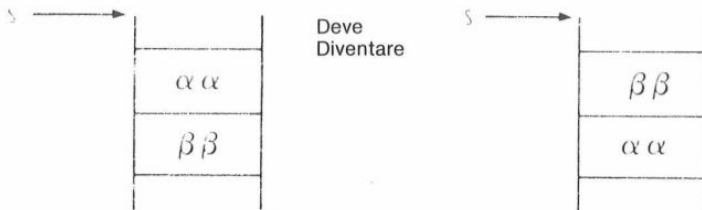
Esercizio 3.24 *Scrivere la sequenza di salvataggio di contesto e quella di ripristino (si mette in pila nell'ordine P S X Y).*

La pila può anche servire a trasmettere parametri fra sottoprogrammi. Ciò è poco usato nel 6502 che ha, con l'indirizzamento indiretto, un sistema più efficace.
Per mettere in pila S stesso, occorre passare attraverso X grazie a **TSX** e **TXS**. Per inizializzare S, bisogna passare da X poiché "LDS" non esiste sul 6502: per esempio:

```
LDX #0$FF
TXS
```

La trasmissione tra S e X combinata con l'indirizzamento indicizzato permette di esaminare – senza togliere dalla pila – qualsiasi elemento della pila:

```
TSX
LDA    $101,X   ;  sommità della pila
....
INX
LDA    $101,X   ;  elemento precedente.
```

Esercizio 3.25 *Scambiare di posto nella pila la sommità e l'elemento precedente:*

Esercizio 3.26 *Come nell'esercizio 3.5 si vuole trasferire il contenuto di X in Y, ma senza perdere il contenuto di A.*

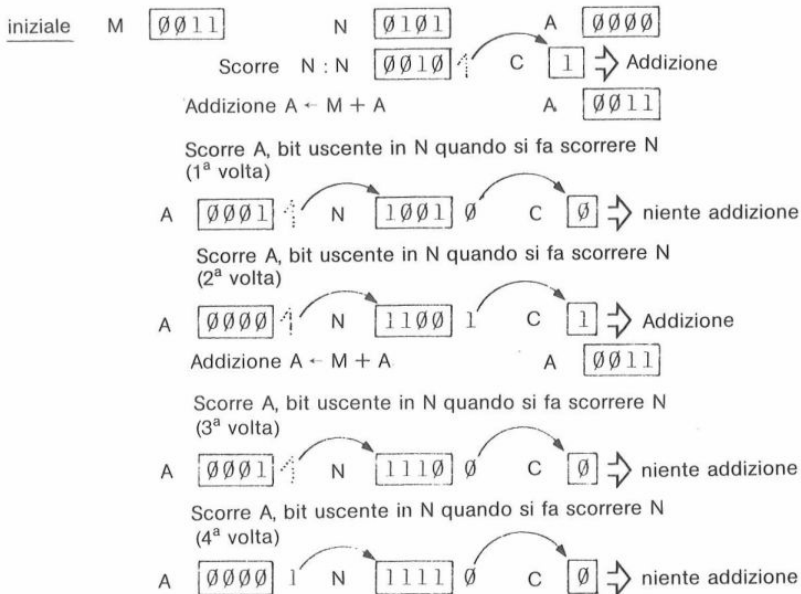
RIEPILOGO

Abbiamo visto ora le principali istruzioni e i principali modi di indirizzamento, così come esempi della loro utilizzazione. Il modo di indirizzamento più elaborato è l'indirizzamento indiretto indicizzato che il 6502 è praticamente l'unico ad offrire sul mercato: abbiamo visto quanto faciliti la manipolazione di array.
La tecnica degli array è con quella dei sottoprogrammi, una delle più importanti studiate in questo capitolo. Entrambe sono in atto nel nostro programma di disegno animato, applicazione in cui il ricorso al linguaggio macchina è necessario per ottenere una velocità sufficiente. Rivedete anche il programma di ordinamento citato alla fine del capitolo 1.
Non possiamo terminare questo capitolo senza fare un'applicazione che permetterà di rivedere come sfruttare al meglio le risorse di un microprocessore.

MOLTIPLICAZIONE 8 BIT PER 8 BIT

Desideriamo moltiplicare due numeri di 8 bit (senza segno) M ed N situati rispettivamente agli indirizzi $22 e $23. Il risultato sarà di 16 bit.
L'algoritmo è esattamente lo stesso che in decimale: sia R la memoria (2 byte) che conterrà il risultato e sia inizialmente nulla.
Se il bit più a destra di **N** è 1, si aggiunge **M** a R (=0). Poi si fa scorrere la finestra di addizione di un posto a sinistra. Se il bit di rango corrispondente di **N** è 1, si aggiunge **M** a **R** nella finestra di addizione in cui si è. Se il bit di N è nullo, non c'è addizione.

Dopo avere fatto 8 scorrimenti, la moltiplicazione è terminata.
Nel nostro caso, la finestra di addizione sarà l'accumulatore A (unico registro in cui si possano eseguire addizioni).
Alla fine della moltiplicazione, conterrà il byte alto di **R**. Invece di far scorrere la finestra di addizione verso sinistra, facciamo scorrere verso destra R, che fa lo stesso.

Chiamiamo **B** provvisoriamente la memoria che riceve mano a mano i bit uscenti a destra di A.
Per testare i bit successivi di **N**, basta far scorrere N verso destra: ogni bit cade allora nel flag **C** in cui è immediatamente testabile. Ma allora, poiché facciamo scorrere **N** verso destra, si crea spazio per ricevere un bit uscente da A: dunque **B** ed **N** possono essere sovrapposti.
La **figura** qui sotto mostra gli stati successivi durante la moltiplicazione M=0011. N=0101 (facciamo il caso di 4 bit · 4 bit per semplificare).

Ci si ferma perché ci sono stati quattro scorrimenti A,N.

Esercizio 3.27 *Redigete lo stesso schema per M=1111 e N=1111.*

Pensiamo che questo esercizio abbia sufficientemente esplicitato l'algoritmo. Vi consigliamo tuttavia di non esitare a stendere il flowchart.
Sfrutteremo ora le proprietà dello scorrimento del 6502. Lo scorrimento iniziale di N è un semplice LSR.
Come far scorrere A,N in modo che il bit uscente alla destra di **A** entri a sinistra di **N**? Ebbene, prima di tutto scorrimento a destra di **A**, il bit uscente è nel flag di riporto. Ma quest'ultimo entra a sinistra di **N** se facciamo **ROR N** (vedere schema pag. 20).

D'altronde, anche lo scorrimento di **A** deve essere una **ROR** perché l'esercizio 3.27 ha mostrato che il riporto risultante dall'addizione deve entrare a sinistra di **A**.
Da cui il programma, sapendo che X è usato per contare le 8 iterazioni:

```
0000          M       = $22
0000          N       = $23
0000          *       = $C000
C000 A9 00    MUL     LDA #0
C002 A2 08            LDX #8      ; INIZIALIZZAZIONE
C004 46 23            LSR N       ; CONTATORE
C006 90 03    CICLO   BCC NONAD   ; SCORRIMENTO INIZIALE
C008 18               CLC         ; C<>0 -> ADD
C009 65 22            ADC M       ; C <- 0 PER
C00B 6A       NONAD   ROR A       ; ADDIZIONE
C00C 66 23            ROR N       ; SCORRIMENTO
C00E CA               DEX         ; A,N
C00F D0 F5            BNE CICLO   ; C NON MODIFICATO
C011 60               RTS
```

Abbiamo costituito il programma come sottoprogramma, ma voi sostituite RTS con BRK (00) per le prove.
Per esempio, se mettete 64 esa (=100) in 1 e 2, dovreste trovare 2710 esa in A,N. Se mettete FF in 1 e 2, dovreste trovare FE01 in A,N.

Esercizio 3.28 *Quale sarebbe la pecca del programma se ROR A fosse sostituito da LSR A?*

Vi suggeriamo di tentare $16 \cdot 16 \rightarrow 32$ ed anche la divisione...
Abbiamo ora terminato questa esplorazione del set di istruzioni del 6502. Siamo giunti col disegno animato al limite dell'assemblaggio a mano ([1]).
Per poter trattare problemi più complicati, bisogna munirsi di un assembler. Il breve capitolo seguente ve ne spiegherà il modo d'uso.

([1]) In effetti, i listati di questo capitolo sono stati ottenuti con il macroassembler $8000, $C000.

Capitolo 4

Utilizzo dell'assembler

L'assembler è un programma che traduce in linguaggio macchina il testo che avete scritto in assembler simbolico.
Insomma, si incarica del compito fastidioso che noi abbiamo finora fatto a mano: riconoscere i codici mnemonici e il modo di indirizzamento, cercare nell'appendice II, scrivere il codice operativo esadecimale, tradurre gli indirizzi degli operandi, ecc.
Descriviamo, qui, l'utilizzo dell'Editor Assembler universale EDASM. Esistono altri assembler sul mercato ed anche un disco ufficiale Commodore.
In ogni caso, i principi di utilizzo sono esattamente gli stessi: solo qualche dettaglio nella presentazione e la scrittura di direttive possono essere diversi da un assembler all'altro.

SVILUPPO DI UN PROGRAMMA

Lo sviluppo di un programma comprende quattro passi principali (oltre il passo preliminare che consiste nel concepire il programma!).

Caricamento del programma simbolico

Si chiama anche programma sorgente. Questa tappa è fatta con l'aiuto **dell'editor** EDASM che vi permette di scrivere delle linee, di fare correzioni ed inserimenti. Una volta costituito il testo-sorgente nel modo che desiderate, avete la possibilità di salvarlo su disco o cassetta (comando S) per richiamarlo successivamente (comando C).

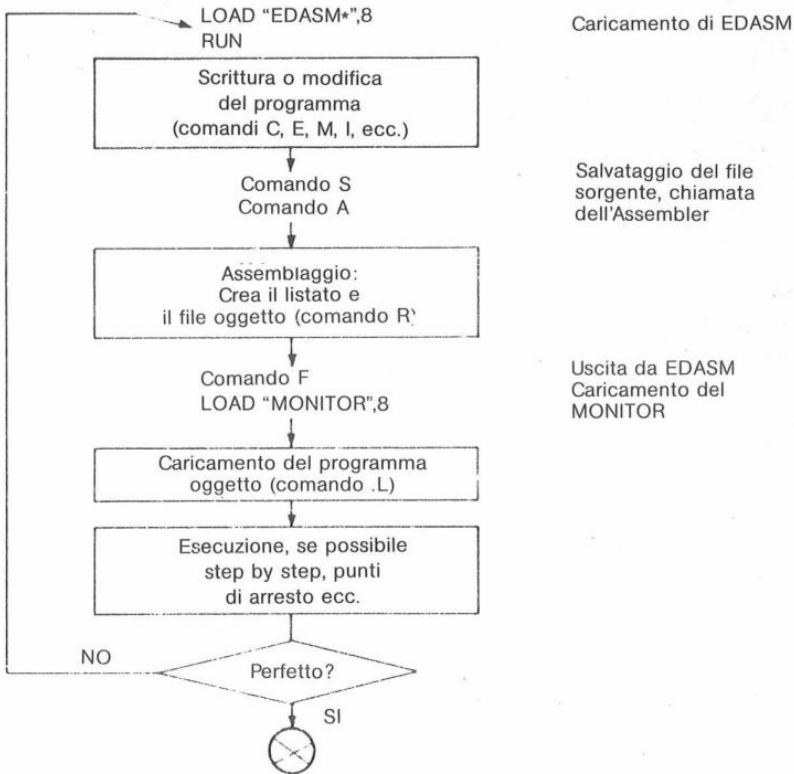
Assemblaggio propriamente detto

Il programma assemblato (lo si chiama anche "programma oggetto") è installato direttamente da EDASM nel posto in cui deve risiedere in memoria. In più, avete la possibilità di salvare il programma-oggetto su disco o cassetta (comando **R**). Inoltre, il programma è listato, a scelta, sul video o sulla stampante.

Caricamento

È il caricamento del programma oggetto nella zona di memoria in cui verrà eseguito. Questo caricamento non è da fare se avete appena fatto l'assemblaggio con EDASM. Se, al contrario, avete interrotto la vostra sessione di lavoro, dovete ricaricare il file creato alla fine dell'assemblaggio col comando **R**. Ciò è fattibile col comando Basic diretto LOAD "nome",1 o 8,1 o col comando .L del monitor che voi utilizzate. Notate che se utilizzate SUPERMON immediatamente dopo l'assemblaggio e se il vostro programma-oggetto è installato nella parte alta della memoria (per esempio C000...), il caricamento è inutile: il caricamento di SUPERMON non cancella il vostro programma oggetto.

Esecuzione

È la parte più delicata e non è sufficiente fare SYS **indirizzo** (o .G indirizzo col monitor). Infatti, in caso di errore o di ciclo infinito, non sperate di riprendere il controllo della macchina col tasto STOP.
La combinazione STOP/RESTORE vi permetterà probabilmente di riprendere il controllo, salvo casi molto rari in cui il vostro errore ha comportato modifiche importanti in memoria, riotterrete il cursore e – al contrario di RESET – la memoria non sarà rimessa a zero.
Un'altra soluzione è di utilizzare un monitor molto perfezionato che possieda tutte le funzioni del monitor linguaggio macchina che abbiamo utilizzato al capitolo precedente, più qualche altra molto utile per la messa a punto del programma come l'esecuzione step by step e la possibilità di punti di arresto.
Quando avrete trovato degli errori, dovrete correggere il programma sorgente ricaricando EDASM e il file sorgente.
Fate le vostre correzioni, poi riassemblate e così di seguito.
Questa successione delle operazioni è esplicitata nel diagramma a pagina seguente (si è supposto che disponiate del disco, altrimenti sostituite gli 8 con 1).
Vediamo ora le modalità d'uso in modo più dettagliato di ciascun programma.

Flow chart della messa a punto di un programma

USO DELL'EDITOR

Dopo il caricamento e RUN, EDASM visualizza un menu: otterrete la funzione desiderata premendo la prima lettera (del corrispondente in inglese).

- ***E***: *scrittura di un programma*
- ***M***: *modifica*
- ***I*** : *inserimento di una linea*
- ***D***: *cancellazione di una serie di linee*
- ***L*** : *listing*
- ***A***: *assemblaggio*
- ***C***: *caricamento file sorgente*
- ***S***: *salvataggio file sorgente*
- ***R***: *memorizzazione file oggetto*
- ***F***: *fine (uscita da EDASM)*

I comandi concernenti l'editazione sono ***E, M, I, D, L, C, S,*** e ***F***. EDASM funziona in modo minuscolo.

- ***E***(in verità "e" poiché siamo in modo minuscolo)
  comando di scrittura di un programma

Il sistema vi domanda il numero di linea di partenza. Dovete rispondere 1 se iniziate a scrivere un programma, altrimenti potete riprendere la scrittura in qualsiasi numero di linea. Se una linea del numero indicato esiste già, viene listata e sarà sostituita da quella che scriverete.
Il sistema procede in seguito alla numerazione automatica. Le linee preesistenti che voi riscrivete saranno sostituite. Le altre vengono conservate. Terminate la scrittura premendo 'f' come prima lettera di una linea.
La scrittura di una linea si fa fornendo in successione i quattro campi dell'istruzione: label, mnemonico, operando, commento. Si passa da un campo al seguente quando si fa uno spazio o si è raggiunto il numero massimo di caratteri permessi per tale campo (6 per una label, 4 per uno mnemonico, 10 per l'operando, 11 per il commento). Se avete allungato dei campi per inavvertenza o se vi accorgete di un errore nel campo precedente, potete tornare indietro con "←" (freccia a sinistra). Si corregge nel campo attuale con l'aiuto di **DEL**. Infine, si può tornare alla linea precedente con " ↑ " all'inizio di una linea.
Si passa alla linea successiva sia facendo Return che raggiungendo il limite di spazio disponibile del campo commento.

- ***M: modifica di una linea***

Per modificare tutta la linea, è meglio riscriverla col comando **E**. Altrimenti il comando **M** vi chiede il numero di linea da modificare. La linea viene listata. Vi viene domandato quale campo desideriate modificare (l: label, m: mnemonico, o: operando, c: commento). Riscrivete tutti i campi sulla linea.

- ***I: inserimento di una linea***

Vi viene domandato prima di quale linea si fa l'inserimento e quante sono le linee da inserire.

- ***D: cancellazione di una serie di linee***

Vi viene domandato il numero della prima linea da cancellare, poi il numero prima del quale si fermerà la cancellazione: per quest'ultimo numero, bisogna dunque rispondere 'ultima linea da cancellare +1'.

Esempio

Per cancellare le linee 3, 4, 5 e 6, rispondereste 3 e 7.

• L: listato del programma sorgente attualmente in memoria

Vi viene domandato il numero della prima linea da listare (default=1) e quello dell'ultima (default=ultima esistente). Per listare tutto il programma basta fare **L** Return Return.

• C: caricamento di un programma sorgente

Vi viene domandato il dispositivo (1=cassetta, 8 o 9=disco) e il nome del file. Esempio di risposta:

8, MIOPROG-S
1, TIZIO-S

• S: salvataggio del programma sorgente

Rispondete alle stesse domande di C. È consigliabile apporre un suffisso -S che vi ricordi che è un file sorgente.

• F: uscita dal programma

Nella versione interpretata, si può ritornare al programma senza perdita di testo facendo GOTO 250.

UTILIZZO DELL'ASSEMBLER

L'assemblaggio è lanciato dal comando **A**.
L'assembler procede in due passate: il testo sorgente è letto due volte. Nel corso della prima passata, il testo è esplorato e viene preparata la symbol table (tabella dei simboli). Il codice in linguaggio macchina è generato durante la seconda passata.
Ottenete infine il listing del testo sorgente e codice generato con gli eventuali messaggi di errore.
Diamo, di seguito, la lista di **quattro possibili messaggi di errore** con la loro interpretazione. È indicato il numero della linea in cui si trova l'errore.

Salto condizionato distanza troppo lunga linea...

Salto con indirizzamento relativo su una distanza troppo elevata.
Il salto relativo può essere fatto solo se lo spiazzamento con segno è di un byte, quindi a +/–127 byte di distanza. Se l'assembler constata che questa condizione non è verificata segnala questo errore. Ciò può avvenire quando, dopo la modifica di un programma, si aggiungono molte istruzioni tra l'istruzione di salto e il suo punto di arrivo: la condizione era soddisfatta prima ma non più dopo. Il rimedio è di utilizzare una staffetta oppure un JMP: per esempio:

```
BNE LONTANO
```

sarà sostituita da:

```
BEQ PROSEGUI
JMP LONTANO
PROSEGUI...
```

Mnemonico illegale linea...

Codice operativo illegale o assente. Ciò può accadere sia se il codice mnemonico è scritto scorrettamente, sia se, per un'istruzione che non ha label, avete dimenticato di passare al campo seguente: il codice mnemonico è preso allora come label e la macchina prende il campo seguente come codice operativo.

Modo di indirizzamento illegale linea...

Uso di un modo di indirizzamento non autorizzato dall'istruzione considerata, o errore di scrittura nell'operando. Avete utilizzato l'accumulatore **A** come operando in un'istruzione diversa da ASL, LSR, ROL o ROR, o avete usato A come nome di variabile, il che è proibito.

Operazione simbolica non definita linea...

Si è fatto riferimento a un simbolo che non è definito da nessuna parte, cioè non appare né come label né a sinistra di una direttiva =.

Sintassi

L'assembler EDASM utilizza la sintassi standard dell'industria per i modi di indirizzamento, così come l'abbiamo vista al capitolo 3. Permette di designare gli indirizzi o i dati nella forma:

- costanti: esempio $E84C (i prefissi possibili sono niente per il decimale, $ per l'esadecimale, % per il binario);
- simboli: esempio CAIO;
- espressioni.

È il punto sul quale EDASM differisce maggiormente dagli standard dell'industria. Al posto di scrivere, per esempio:

```
LDA CAIO+1
```

per dire che si vuole caricare il byte successivo a CAIO, potete scrivere in EDASM:

```
LDA CAIO+
```

Allo stesso modo, al posto di:

LDA TIZIO–2

scrivete:

LDA TIZIO––

+ e – sono i soli operatori permessi (i soli utili in pratica) e il numero di segni permessi è definito dal numero limite di caratteri del campo operando.

– stringhe di caratteri: il prefisso è '. Solo il primo carattere è preso in considerazione.

Esempio

LDA # 'A fa la stessa cosa di LDA #$41
LDA # 'ABCD avrebbe lo stesso effetto.

è il simbolo di indirizzamento immediato. Il # può essere seguito da una costante (#$FF), da un simbolo (#CAIO), o da < oppure > che designano rispettivamente il byte basso e il byte alto dell'indirizzo che segue (esempio se TIZIO=$CDEF , LDA #>TIZIO carica $CD in accumulatore).

Direttive di assembler

Esaminiamo ora le direttive offerte dall'assembler EDASM. Le abbiamo già viste in parte nel capitolo precedente. Ce ne sono due ed hanno il ruolo di definizione dati.

= : assegna un valore (indirizzo o dato) ad un simbolo.

Esempio

PCR = 12
VIA = $E840

A destra del segno = può esserci un simbolo, ma quest'ultimo deve essere stato definito precedentemente (nessun riferimento in avanti). Trarrete grande vantaggio nel parametrizzare al massimo i vostri programmi e a definire i vostri parametri in testa al programma con delle direttive =. In caso di modifica, dovrete modificare solo qualche istruzione.

= ammette il seguente caso particolare:

*=: il simbolo * designa il "contatore di allocazioni" che, ad ogni istante, contiene l'indirizzo a cui l'assembler è pronto ad installare la prossima istruzione assemblata. Insomma * corrisponde durante l'assemblaggio, a ciò che sarà PC durante l'esecuzione.

Quindi *= MASP, significa assegnare a * il valore MASP, cioè installare la prossima istruzione a partire dall'indirizzo MASP; è la direttiva di origine.

Esempio

* = $C000

(su alcuni altri assembler si scrive ORG $C000).

BYT : riserva un byte e vi piazza un valore iniziale.

Esempio

QUI BYT 2

installa il valore 2 nel byte di indirizzo QUI.
Se si vuole semplicemente riservare un byte, basta metterci un valore qualunque.

CARICAMENTO DEL PROGRAMMA

È prudente, dopo l'assemblaggio, salvare il programma oggetto col comando **R**.
Vi si domanda, come per S, il dispositivo e il nome del file.

Esempio di risposta:

8, PROG-O

Si consiglia di apporre un suffisso –O che vi ricorda che si tratta di un programma oggetto.
In caso di bisogno il programma si ricarica con:

- LOAD "nome", dispositivo,1 (,1 è indispensabile per evitare traslazioni);
- o col comando .**L** del monitor. Supponiamo qui che stiate usando SUPERMON.

SUPERMON contiene tutti i comandi **G, L, M, R, S** e **X** che abbiamo descritto al capitolo 2.
Ma contiene anche comandi supplementari molto utili nella messa a punto del programma e che ora andiamo a descrivere.

Facilitazioni nella scrittura e nell'esame di programmi

.A: Mini assembler

Si tratta di un mini assembler che traduce il codice mnemonico ma non tratta i nomi di variabile. Dovete quindi scrivere un'istruzione

nella forma per esempio: LDA $2000. Tutti gli indirizzi e i dati devono essere forniti in esadecimale, con quattro cifre (esempio $0F00 e non $F00) per gli indirizzi di 16 bit e due cifre per i dati di 8 bit.
Se fornite un indirizzo della pagina zero su due cifre sarà assemblato con indirizzamento pagina zero.

Esempio

LDA $F0 darà A5 F0.

Se lo darete con quattro cifre, sarà assemblato in indirizzamento diretto (LDA $00F0 darà AD F0 00).
Per lanciare l'esecuzione del mini assembler, premete A l'indirizzo di partenza e la vostra prima istruzione.

Esempio

.A 2000 LDA=$00 (il punto è dato da SUPERMON).

Quando premete 'Return', la linea diventa:

.A 2000 A9 00 LDA=$00

e la linea seguente è .A 2002 ■ (cursore) allora dovete soltanto scrivere l'istruzione seguente, e così di seguito. Terminate premendo 'Return' tout court all'ultima domanda di istruzione.
Ma, nel medesimo tempo, SUPERMON carica il testo assemblato agli indirizzi specificati.

Esercizio di allenamento 4.1 *Caricate SUPERMON (o il vostro monitor) e fatelo partire. Premete:*

.A 2000 LDA $0F

Lo schermo mostra:

```
.A 2000 A5 0F    LDA    $0F
.A 2002
```

Premete LDX $00F, *poi* LDY $000A, *poi* 'Return'.
Dovreste vedere:

```
.A 2000 A5 0F       LDA  $0F
.A 2002 AE 0F 00    LDX  $000F
.A 2005 AC 0A 00    LDY  $000A
.
```

Verifichiamo il contenuto della cella di memoria 2000 e seguenti con M 2000-2008. Dovreste trovare:

```
.:2000 A9  0F AE 0F 00 AC 0A 00
.:2008 AA  AA - - - - - - - - - - - - - AA
```

Ma, in più, il sistema funziona come editor sullo schermo. Se tornate sulla linea LDA $0F e volete sostituire la A di LDA con X, quando premete 'Return' otterrete:

.A 2000 A6 0F LDS $0F

E potrete verificare con .M che in 2000 c'è A6.

Domanda: cosa succede se la modifica varia lo spazio di memoria occupato dall'istruzione?
Il sistema ne esce molto bene: quando premete 'Return' sulla linea modificata, il cambiamento è effettuato poi la linea seguente diventa:
.A nuovo indirizzo, vecchia istruzione
Se voi premete 'Return', l'istruzione è ora caricata nel suo nuovo indirizzo. Quindi, quando un tale cambiamento ha luogo, occorre fare 'Return' su tutte le linee che seguono affinché il cambiamento si propaghi.

Esercizio di allenamento 4.2 *Con la visualizzazione risultante dall'esercizio 4.1, tornate alla linea LDX $0F (la prima) e trasformatela in LDA $000F. Fate 'Return'. Diventa:*

.A 2000 AD 0F 00 LDA $000F

e la seguente è:

.A 2003 AE 0F 00 LDX $000F

Non fate 'Return'. Col movimento del cursore spostatevi su .M 2000-2008. Fate 'Return' per avere la visualizzazione della memoria:

.:2000 AD 0F 00 0F 00 AC 0F 00

Si vede, in 2002, che 00 ha sostituito AE che c'era prima. L'istruzione LDX è stata demolita.
Risalite alla linea LDX e fate 'Return' due volte. Ritornate su .M poi fate 'Return'. Questa volta, il programma è stato interamente rettificato e vedrete:

.:2000 AD 0F 00 AE 0F 00 AC 0F
.:2008 AA ------------------ AA

Esercizio di revisione 4.3 *Trasformate la seconda linea in LDX #0F.*

Si vede, in conclusione, che questo *mini assembler è molto pratico.* Calcola lo spiazzamento per le biforcazioni, ma bisogna fornirgli esplicitamente l'indirizzo a cui saltare.

.D: Disassembler

.D 2000-2008 disassembla (cioè traduce l'esadecimale in codice mnemonico) a partire dall'indirizzo 2000.

Si riempie una pagina di schermo.
Anche qui, il sistema funziona con l'editor di schermo su una linea disassemblata, per esempio:

.,2000 AD 0F 00   LDA $000F

cambiate AD in AE, la linea diverrà:

.,2000 AE 0F 00   LDX $000F

e la memoria sarà modificata.
Potete anche passare dal disassembler all'assembler e viceversa premendo ',' dietro il punto per avere il disassembler ed allora potete modificare il testo esadecimale, oppure A dietro il punto per avere l'assembler ed allora potete modificare il codice mnemonico.

.P: Disassemblaggio continuo

Qui si disassembla in continuo, il che è comodo per un disassemblaggio su stampante. Bisogna fare allora OPEN 4,4:CMD 4 prima di eseguire il disassemblaggio.
Ecco, ora, alcuni comandi di aiuto alla scrittura meno spettacolari.

.F: Riempimento memoria

.F 2000-3000 FF riempie tutti i byte di indirizzo compreso tra 2000 e 3000 col valore FF.
Ciò è utile per inizializzare un array.

.H: Ricerca di una sequenza di byte in memoria

.H C000 D000 20 D2 FF ricerca tra gli indirizzi C000 e D000 la serie 20 D2 FF (JSR $FFD2) e visualizza gli indirizzi trovati.
Si può cercare una sequenza con un massimo di 32 byte. La sequenza può essere fornita sotto forma di stringa di caratteri: si cerca allora il codice ASCII:

.H A000 FFFF 'ERROR'

ricerca la stringa ERROR fra A000 e FFFF (localizza le locazioni in cui sono implementati i messaggi di errore dell'interpretazione Basic).
Ecco, ora, un comando di trasferimento di una zona di memoria in un'altra.

.T: Trasferimento di memoria

.T 2000 2500 3000 trasferisce i dati degli indirizzi fra 2000 e 2500 nella zona di memoria che inizia a 3000 ((2000) va in (3000), (2001) va in (3001) ecc.). Si tratta di un trasferimento puro e semplice. Per trasferire un programma, bisogna adattare gli indirizzi di salto.

CONCLUSIONE - ALTRO SOFTWARE

Abbiamo, ora, in mano dei programmi di utilità che ci permetteranno di preparare efficacemente i nostri programmi in linguaggio macchina.
Esistono altri programmi al di fuori di quelli da noi descritti. A livello del monitor, alcuni hanno comandi di esecuzione step by step utilissimi per la messa a punto. Riferitevi, per l'utilizzo, alle istruzioni che vi sono accluse.
Ce ne sono anche a livello di assembler, EDASM che abbiamo qui descritto, è nella sua versione compilata, difficilmente superabile nel rapporto prestazioni/prezzo.
In ogni caso, questi sistemi si usano tutti in modo molto simile a quello che abbiamo descritto. La loro prova non mancherà su "La Commode" il che vi permetterà di fare la vostra scelta.

Capitolo 5

Interazioni col Basic

Sappiamo ora preparare programmi o sottoprogrammi in linguaggio macchina e li sappiamo caricare nel C64.
Ciò può essere fatto in modo totalmente indipendente e l'esecuzione può essere comandata sotto il controllo del monitor con **G xxxx**.
Ma, più spesso, ciò che si scrive è un insieme di sottoprogrammi in linguaggio macchina destinati ad essere eseguiti in un **contesto Basic**: spesso si tratta di sostituire alcuni moduli critici che sarebbero troppo lenti in Basic rispetto ai loro equivalenti in linguaggio macchina.
Questa esecuzione, sotto controllo del Basic, pone diversi problemi che non ci sarebbero in un contesto autonomo.

- Come chiamare un sottoprogramma in linguaggio macchina a partire dal Basic? Vedremo che il Basic offre due istruzioni per questo. A tale questione si collega quella della trasmissione di dati tra il vostro programma Basic e il sottoprogramma in linguaggio macchina.
- Dove mettere il programma in memoria? Infatti il Basic occupa posto in memoria. Il suo funzionamento rischia di perturbare i vostri moduli in linguaggio macchina. Vedremo una soluzione per evitarlo.
- Come caricare il programma in linguaggio macchina in relazione al Basic. Vedremo diverse soluzioni: il monitor ne offre una, comoda quando è in ROM. Ma ve ne sono altre.
- Infine, c'è un certo numero di funzioni base, come la visualizzazione sullo schermo, di cui si sa già che sono preprogrammate in ROM. Sarebbe utile sapere come richiamarle a partire da un sottoprogramma in linguaggio macchina. Daremo, in questo capitolo, un certo numero di indirizzi e di esempi di alcuni sottoprogrammi del sistema operativo del C64 in ROM. Ma, qui, ci limiteremo ad una selezione, lasciando al prossimo libro di questa serie (volume III: il sistema operativo del C64), cura di darvi nozioni più complete.

CHIAMATA DI UN S.P. IN L.M. DAL BASIC

Supponiamo di essere riusciti a mettere in memoria, a partire dall'indirizzo IND un sottoprogramma in linguaggio macchina. Il Basic offre due istruzioni che permettono di farlo eseguire a partire da un programma in Basic:

SYS IND che lancia l'esecuzione del codice macchina che comincia all'indirizzo IND. L'indirizzo può essere espresso come costante (in decimale), di variabile, o anche di espressione aritmetica:

SYS 2000
SYS 3·A+4

Esercizio 5.1 *Fate eseguire, a partire dal Basic, il sottoprogramma in linguaggio macchina, che inizia in $C00.*

Domanda: la variabile utilizzata può essere un intero (SYS X%)?

Certo. Ciò obbliga ad usare indirizzi ≤ 32767, il che non spaventa per un programma in RAM. Al contrario, poiché SYS permette di richiamare un sottoprogramma che fa parte del sistema operativo in ROM, ciò non è possibile nella forma SYS X%.
L'istruzione SYS ha delle limitazioni:

- l'indirizzo deve essere intero, inferiore a 65535 (altrimenti VALUE ERROR);
- all'indirizzo indicato, deve cominciare un programma in linguaggio macchina: in particolare, il byte trovato a questo indirizzo deve essere un codice operativo valido;
- la sequenza di istruzioni trovate a partire dall'indirizzo indicato deve terminare in capo a un tempo finito con **RTS** (60 esa = **96 dec**): c'è allora ritorno al programma Basic in seguito ad una SYS che si comporta come un GOSUB oppure, a rigore, con **BRK (00)** e allora si passa in monitor. *Altrimenti si ha bloccaggio integrale della macchina e potete soltanto spegnere e riaccendere.*

La seconda istruzione Basic che permette di eseguire un sottoprogramma in linguaggio macchina è la chiamata della funzione USR nella forma **Y=USR(X)** a condizione d'aver precedentemente messo negli indirizzi 1 e 2 della memoria, l'indirizzo a cui comincia il vostro sottoprogramma in linguaggio macchina.
Si avrà dunque la chiamata seguente se IND è l'indirizzo di inizio del sottoprogramma in linguaggio macchina:

```
100  HIND=INT(IND/256)     : REM byte alto dell'indirizzo
110  LIND=IND–256*HIND     : REM byte basso dell'indirizzo
120  POKE 785,LIND         : POKE 786,HIND
130  Y=USR(X)
```

Si sa che l'istruzione POKE permette di scrivere in memoria. È ciò che permette di scrivere nell'indirizzo 1 della memoria, il byte basso dell'indirizzo del sottoprogramma e nell'indirizzo 2, il byte alto (linea 120).

Domanda: cosa succede se si dimentica di mettere i valori giusti in 785 e 786?

Certamente non sarà il vostro sottoprogramma in linguaggio macchina ad essere eseguito! Rischiate di saltare ad un indirizzo che non significa nulla e quindi di bloccare il sistema, naturalmente se avete usato gli indirizzi 785 e 786 per altre cose precedentemente, all'inizializzazione del sistema contengono l'indirizzo di una routine di errore.

Esercizio 5.2 *Saltare a $C000 con USR.*

Passaggio di parametri

Non ci resta, ora, che spiegare il ruolo di **X** e **Y** in **Y=USR(X).**
Ciò giustificherà che, in alcuni casi, si usi USR piuttosto di SYS che è più semplice.
USR permette un certo passaggio di parametri (cioè di informazioni) tra il programma chiamante e il sottoprogramma in linguaggio macchina.
Mentre SYS effettua un semplice salto all'indirizzo indicato, USR effettua la serie seguente:

1. Calcola l'espressione che è in X e mette il risultato nell'accumulatore mobile.
2. Salta all'indirizzo contenuto in 785 e 786.

Dopo il ritorno dal sottoprogramma in linguaggio macchina, si mette, nella variabile Y, il contenuto dell'accumulatore mobile.
Naturalmente, sta al vostro sottoprogramma in linguaggio macchina recuperare il risultato dell'espressione nell'accumulatore mobile, prima di RTS, mettere nell'accumulatore mobile le informazioni che vorreste trovare in Y alla fine.
L'accumulatore mobile è agli indirizzi $61-$66, seguendo la struttura che è stata descritta al capitolo 2 (esponente, mantissa, mantissa, mantissa, mantissa, segno).
L'inconveniente di questo modo di passare i dati è che permette un solo parametro inviato al sottoprogramma in linguaggio macchina e un solo parametro ritornato. Se si volessero passare più informazioni, ci sono altri modi che, d'altronde, sono utilizzabili anche con SYS.

1. Una maniera semplicissima di passare un byte è di convenire un indirizzo nel quale memorizzarlo.

Per esempio, se decidiamo di inviare un byte al sottoprogramma in linguaggio macchina passando per l'indirizzo 5000 ($1388), la sequenza di chiamata sarà:

```
POKE 5000,dato : SYS INDSP
```

Nel sottoprogramma in linguaggio macchina, il dato sarà recuperato, per esempio, con:

```
AD 88 13    LDA $1388
```

Ugualmente, il sottoprogramma in linguaggio macchina può produrre un risultato che si conviene piazzare all'indirizzo 8000 ($1F40): se il risultato è generato nel registro X, si avrà:

```
8E 40 1F STX $1F40
60       RTS
```

e proprio dopo la SYS, sarà sufficiente fare: X=PEEK(8000).
Si può, in questo modo, scambiare numerosi dati a condizione di scegliere indirizzi "tranquilli". Questo problema viene trattato nella sezione seguente.

2. Si sa come il Basic memorizzi le proprie variabili. Quindi se si vuole, per esempio, che il sottoprogramma in linguaggio macchina agisca sulla variabile A, basta cercare la variabile A nella zona delle variabili. Come sono memorizzate le variabili?

Ebbene, ogni variabile occupa 7 byte nella zona di memoria che segue la fine del testo in Basic. Poiché la frontiera è mobile, l'inizio di tale zona è indicato da un puntatore agli indirizzi 45 e 46 ($2D e $2E).
I due primi byte dei sette sono il codice ASCII dei primi due caratteri del nome della variabile, così come per una variabile reale, + 80 esa per il secondo carattere per una variabile intera, + 80 esa per entrambi i caratteri se si tratta di una variabile stringa. Se il nome ha un solo carattere, il codice del secondo è 0 (o 80).
La fine della zona di variabili è puntata dagli indirizzi 47 e 48 ($2F e $30). Di conseguenza per trovare una variabile, è semplice: si guarda il byte puntato da 45 a 46 e il seguente. Si confronta con il nome cercato. Se non c'è coincidenza, si aggiunge 7 e così di seguito. Ci si ferma se si trova la variabile o si raggiunge l'indirizzo contenuto in 47 e 48: in quest'ultimo caso, la variabile cercata non esiste ancora.

Esercizio 5.3 *Ottenete in 254 e 255 l'indirizzo della variabile TOT (se non esiste 0 e 0).*

Il programma potrebbe d'altronde abbastanza semplicemente essere modificato per recuperare la posizione della variabile fra tutte le variabili.

Ciò che è più interessante, è che un tale programma è inutile. È sufficiente che voi lasciate lavorare per voi il sistema. Infatti, agli indirizzi 71 e 72 ($47 e $48) si trova un puntatore alla variabile corrente.
Se volete l'indirizzo di A, basta scrivere:

```
10 A=....
20 X=A : REM X variabile per salvare A
30 A=PEEK(71)+256*PEEK(72)–2
40 PRINT"L'INDIRIZZO DI A È";A
50 A=X : REM si ristabilisce A
```

il –2 della linea 30 è dovuto al fatto che in 71 e 72 c'è l'indirizzo a cui effettivamente comincia il valore della variabile: col –2 si ottiene l'indirizzo in cui inizia il nome.
Una volta ottenuto l'indirizzo, è facile trasmetterlo al nostro sottoprogramma in linguaggio macchina.

3. È facilissimo trasmettere, a un sottoprogramma in linguaggio macchina, il risultato di un'espressione aritmetica Basic. Se si scrive:

   SYS indirizzo,espressione

   quando il controllo verrà preso dal vostro programma in linguaggio macchina, l'interprete Basic sarà posizionato sulla virgola. È sufficiente allora fare successivamente JSR $AEFD (routine in ROM che verifica la presenza di una virgola), poi JSR $AD8A (routine in ROM che valuta l'espressione aritmetica a cui ci si trova – fino alla prossima virgola – e mette il risultato nell'accumulatore mobile).

Se volete che l'espressione sia intera, potete in seguito fare JSR $B7F7, routine in ROM che converte in intero il numero contenuto nell'accumulatore mobile e mette il risultato:

byte basso in Y, $65 e $14
byte alto in A, $66 e $15

Provate allora il programma seguente che memorizza in 49152, 49153 e 49154, i valori di tre espressioni aritmetiche:

```
10 A=1 : B=2 : C=3
20 SYS 49155,A+B,B·C,A+C–2
30 PRINT PEEK(49152),PEEK(49153),PEEK(49154)
```

sottoprogramma in linguaggio macchina:

```
0000              VIRG  = $AEFD  ; VERIFICA ','
0000              EXPR  = $AD8A  ; VALUTA ESPRESSIONE
0000              INTE  = $B7F7  ; CONVERTI IN INTERO
0000              *     = $C003
C003 20 FD AE           JSR VIRG ; ESPRESSIONE 1
```

segue

seguito

```
C006 20 8A AD        JSR EXPR
C009 20 F7 B7        JSR INTE
C00C A5 14           LDA $14
C00E 8D 00 C0        STA $C000
C011 20 FD AE        JSR VIRG   ; ESPRESSIONE 2
C014 20 8A AD        JSR EXPR
C017 20 F7 B7        JSR INTE
C01A A5 14           LDA $14
C01C 8D 01 C0        STA $C001
C01F 20 FD AE        JSR VIRG   ; ESPRESSIONE 3
C022 20 8A AD        JSR EXPR
C025 20 F7 B7        JSR INTE
C028 A5 14           LDA $14
C02A 8D 02 C0        STA $C002
C02D 60              RTS
```

Si vede come si "fa lavorare" la ROM per recuperare ogni espressione. Vedremo altre routine utili in ROM nella quarta sezione del capitolo.

DOVE METTERE IL VOSTRO SOTTOPROGRAMMA?

Il problema è di trovare una zona di memoria che non venga alterata dal funzionamento del Basic e che, dal canto suo, non disturbi il funzionamento del Basic.
Una soluzione possibile – se non si impiega il registratore – è di utilizzare il buffer di registratore da $33C a $3FB (in effetti da $334 a $3FF). Ma, ci sono due grossi inconvenienti:

1. questa soluzione è applicabile solo se non ci si serve del registratore;
2. la zona offerta non è troppo grande.

Ci sono altre zone tranquille, vista la grande memoria del C64. In particolare, la RAM da $C000 a $CFFF è totalmente al riparo dal Basic; è per questo che l'abbiamo utilizzata finora. Però questa zona non è al riparo dalle estensioni del Basic che si trovano in cartuccia. Queste estensioni si mettono a priori in $8000-$8FFF, ma alcune si ricollocano in $C000 (è proprio ciò che fa la cartuccia IEEE DAMS) oppure vi caricano delle variabili.
In altri contesti, quando si ha una ROM d'estensione da $8000 a $8FFF, la zona $9000-$9FFF diventa tranquillissima.
Abbiamo dunque trovato qualche zona abbastanza vasta per una possibile memorizzazione di una routine in assembler, ma queste zone

sono libere o no, a seconda del contesto in cui ci si trova (Basic o presenza di questa o quella cartuccia).
È per questo che occorrono altre soluzioni.
Un'eccellente soluzione consiste nell'ingannare il C64 con la mediazione dei puntatori che ha in memoria. Agli indirizzi \$37 e \$38 (55,56 dec) figura un puntatore che contiene 1 + ultimo indirizzo RAM disponibile.
Questo puntatore è creato durante l'esplorazione della memoria al momento dell'inizializzazione. Contiene normalmente 00,A0.

Esercizio 5.4 *Verificatelo.*

Ma se per caso, con delle POKE a questi indirizzi, cambiate il valore di questo puntatore, a partire da questo momento, il C64 "*crederà*" che la sua memoria termini all'indirizzo che voi indicate e non andrà mai più a scrivere più in là ciò resta valido sino a un RESET).
Di conseguenza, per riservarvi una zona alla fine della memoria per il vostro sottoprogramma in linguaggio macchina, basta diminuire il contenuto dell'indirizzo 56. Ogni volta che diminuite questo indirizzo di 1, vi riservate una pagina di 256 byte.
Per esempio, POKE 56,128 vi riservate 8 Kbyte alla fine della memoria.
Potreste scrivere POKE 56,PEEK(56)–32.
Questo agisce sulla memoria a blocchi di 256 byte. Si potrebbe essere più precisi agendo anche su 55, ma è spesso inutile.

Esercizio 5.5 *Riservate 384 byte alla fine della memoria del C64.*

Una volta riservata questa zona, potete installarci il vostro programma.
Esiste una terza soluzione che vedremo a proposito del problema del caricamento dei programmi.

Collocazione dei dati

In principio, i dati che potreste essere portati ad utilizzare saranno messi come i programmi nella zona tranquilla che avrete preparato.
Solamente i dati che potete desiderare di installare in pagina zero (è imperativo per un puntatore di indirizzamento indiretto, auspicabile per dati molto usati) pongono un problema particolare.
Col C64 c'è pochissimo spazio disponibile.
Infatti la caratteristica delle nuove versioni del sistema è di utilizzare meglio la pagina zero, il che ha dei vantaggi.
La tabella degli indirizzi dell'appendice 3 segnala alcune celle di memoria della pagina zero che potete utilizzare.

Ciò deve tuttavia essere fatto con prudenza, in funzione del ruolo svolto dagli indirizzi che desiderate utilizzare e delle funzioni del Basic che volete usare contemporaneamente.
Per esempio, se non utilizzate il registratore, potete accedere da $9B a $AB, ecc.
In ogni caso, c'è un metodo radicale utilizzato qualche volta (in particolare in certi programmi venduti): all'inizio, il programma salva da qualche parte in memoria il contenuto della pagina zero e la ripristina alla fine.

Esercizio 5.6 *Scrivete il programma che trasferisce la pagina zero in 7F00-7FFF.*

CARICAMENTO DEI PROGRAMMI

Con un monitor in linguaggio macchina, il salvataggio su cassetta o disco e il caricamento di un modulo in linguaggio non pongono alcun problema: basta usare i comandi S e L di cui la sintassi è stata data al capitolo 4.
Una precauzione: è consigliabile procedere nel seguente modo:

1. caricate il (o i) programma(i) in linguaggio macchina;
2. riservate la zona di memoria voluta (se li avete caricati alla fine della memoria) con POKE 56,...;
3. infine, caricate il vostro programma in Basic.

Questo modo di procedere ha un inconveniente: il caricamento si effettua in due o più volte (è ancora peggio se si deve caricare il monitor).
È per questo che proponiamo, qui, un altro metodo che può essere interessante. Consiste nel mettere il vostro programma in linguaggio macchina proprio dietro il programma Basic e memorizzare i due in blocco su cassetta o disco. Il caricamento si farà allora in una sola volta. Ecco come procedere:

1. Scrivete il programma in Basic.
2. Fate ?PEEK(45)+256·PEEK(46).

Il risultato è l'indirizzo a partire dal quale voi potete installare il vostro sottoprogramma in linguaggio macchina.

3. Scrivete il vostro sottoprogramma in linguaggio macchina e installatelo a quell'indirizzo. È più facile, naturalmente, se avete il monitor in cartuccia. Adattate, nel programma Basic, gli indirizzi di chiamata al sottoprogramma in linguaggio macchina SYS xxxx. Il vostro programma in linguaggio macchina termina ad un certo indirizzo. Sia hh,ll l'indirizzo seguente (primo indirizzo libero) in esadecimale. Convertite in decimale il byte alto (hh) e il byte basso (ll).

4. Fate POKE45,11 :POKE47,11 :POKE49,11
   POKE46,hh :POKE48,hh :POKE50,hh

(questo mette al posto i puntatori di fine del Basic – vedi appendice 3)

5. Fate il SAVE abituale, sia su cassetta che disco.
6. Potete adesso caricare il programma con un solo LOAD normale del Basic.

Questo metodo è molto pratico. Ma c'è qualche precauzione essenziale da prendere, per evitare il bloccaggio:

- Non dimenticate di adattare gli indirizzi delle SYS. Attenzione, l'adattamento non deve modificare il numero di caratteri del programma. Quindi, in partenza, quando non conoscete l'indirizzo in cui sarà l'indirizzo del programma, mettete una chiamata nella forma SYS 0001 (o SYS 00001 se prevedete che l'indirizzo avrà cinque cifre).
- Tutte le modifiche nel programma che cambiano la sua estensione provocano una traslazione del codice macchina, i cui indirizzi saranno allora falsati. Quindi una tale modifica è assolutamente da evitare, oppure in seguito occorre apportare i necessari adattamenti.

Esercizio di allenamento 5.7 *Facciamo questa serie di operazioni su un esempio. Scriveremo un sottoprogramma in linguaggio macchina che avrà per unico effetto di mettere il valore $55 (85 dec) all'indirizzo 2 della memoria. Scriveremo il programma di chiamata in questo modo:*

```
10 A=3
20 SYS0001
30 ?A
40 ?PEEK(2)
```

Le linee 10 e 30 hanno lo scopo di provare che il programma in linguaggio macchina non disturba le variabili Basic. La linea 40 prova che il programma in linguaggio macchina è stato eseguito. Notate l'indirizzo della SYS che sarà modificato.
Fate ?PEEK(45)+256*PEEK(46), otteniamo 2086 ($826) indirizzo di inizio del nostro codice macchina.
Modifichiamo la SYS che diventa SYS2086 e scriviamo il programma in linguaggio macchina:

```
                  * =  $826
0826 A9 55        LDA #$55
0828 85 02        STA $2
082A 60           RTS
```

Noi lo installiamo e annotiamo che il primo indirizzo libero è $82B da cui hh=8 e ll=2B esa = 43 dec.

Facciamo allora le POKE del passo 4. In questo caso, basta fare quelli concernenti ll perché hh non è cambiato. È sufficiente allora fare:

SAVE "TIZIO"
o SAVE "TIZIO",8

In seguito, per verificare che il tutto funziona, fate RESET (spegnete e riaccendete il C64), poi LOAD e RUN. Dovreste vedere sullo schermo 3 e sotto 85.
Per introdurre il programma, senza il monitor, vista la brevità, potete fare:

```
POKE 2086,169 : POKE 2087,85 : POKE 2088,133
POKE 2089,2   : POKE 2090,96
```

Un altro metodo

Un metodo che è stato molto spesso impiegato è il seguente:

1. si scrive il programma in linguagio macchina in decimale sotto forma di DATA;
2. si incorpora, all'inizio del programma, una routine che legga i DATA e installa i valori agli indirizzi desiderati. Per esempio, per il programma semplicissimo appena visto, si avrebbe:

```
10 DATA 169,85,133,2,96
20 FOR I = inizio TO fine
30 READ A : POKE I,A : NEXT
40 SYS inizio
```

Qui, non siamo obbligati a mettere il sottoprogramma proprio alla fine del programma Basic.

Esercizio 5.8 *"Inizio" può essere uguale a 2086?*

Questo metodo ha il vantaggio di essere puramente Basic, ma è molto meno efficiente del precedente. Nel metodo che noi raccomandiamo ogni byte del codice macchina occupa un byte poiché è al suo posto. Nel secondo metodo, ogni byte occupa in media quattro byte! 1 al suo posto più 3 nel DATA (in media due cifre più la virgola). È dunque *terribilmente inefficiente*. Nel primo metodo, il codice macchina non appariva listato mentre nel secondo sì: l'uno o l'altro è preferibile a seconda dei casi (in alcuni problemi di protezione, è meglio che il codice macchina non appaia).

Metodo misto

Infine, si troverà su "La Commode n. 10" un sottoprogramma che permette, a partire dal Basic, di caricare qualsiasi routine in linguag-

gio macchina. Si può quindi ritornare al primo metodo: localizzazione di una zona tranquilla, caricamento in due tempi, ma il caricamento in due tempi non disturba poiché la seconda parte del caricamento è concatenata automaticamente dal programma in Basic che è stato caricato per primo.

UTILIZZO DEI SOTTOPROGRAMMI DI SISTEMA OPERATIVO

Sappiamo ora chiamare, installare e caricare il nostro codice macchina.
Ma, per alcuni problemi comuni, sappiamo che sono risolti nel sistema operativo che è in ROM.
Sarebbe molto vantaggioso poter chiamare le routine corrispondenti; c'è un doppio vantaggio:

- guadagno di spazio in memoria poiché la routine esiste già in ROM; inutile ripeterla in RAM;
- guadagno di programmazione; è inutile rifare un lavoro che è già stato fatto.

Alcune liste di indirizzi sono state pubblicate, segnatamente su "La Commode n. 8". Però, non basta avere gli indirizzi ai quali i vari sottoprogrammi sono installati, occorrono gli indirizzi dei punti di ingresso (verso i quali si può fare JSR). Ma, anche questo non è sufficiente. Occorre anche sapere in quali registri o celle di memoria mettere i parametri di cui la routine ha bisogno e sapere dove questa mette le informazioni che fornisce.
Queste sono le informazioni che forniamo per una serie abbastanza ristretta di routine di sistema operativo.
Vedremo tre categorie di routine: gli ingressi/uscite elementari tastiera/schermo, le routine aritmetiche e la manipolazione di file. Gli indirizzi delle variabili di sistema sono all'appendice 3.

Ingressi/uscite elementari

Potete sempre fare degli output scrivendo direttamente all'indirizzo voluto della memoria di schermo. L'inconveniente è che ciò vi obbliga a fornire il codice schermo di ogni carattere e non il codice ASCII. Le routine di sistema operativo accettano il codice ASCII, cioè eseguono la conversione per noi.
La prima cosa da fare prima di scrivere è di posizionare il cursore, poiché le routine di sistema lo utilizzano. Perciò, due metodi:

1. si carica in \$D1 e \$D2, rispettivamente il byte basso e il byte alto dell'indirizzo della prima cella della linea a cui si vuole andare. Se I è il numero della linea voluta, questo indirizzo è \$0400+40·(I–1)–1. Poi si carica in \$D3 il numero di colonna voluta;
2. si può chiamare la routine di sistema \$FFF0 con:

```
LDX #LINEA–1
LDY #COLONNA–1
CLC
JSR $FFF0
```

Esercizio 5.9 *Posizionare il cursore in mezzo allo schermo; riga 13, colonna 20.*

In seguito, potete scrivere un carattere unico o una stringa.
Per scrivere un carattere, basta caricare il suo codice nell'accumulatore e di chiamare la routine **WRT** all'indirizzo \$FFD2

```
LDA #$41
JSR $FFD2 scrive una A
```

Per scrivere una stringa, si utilizza la routine **STROUT** con punto di ingresso \$AB1E. Questa routine visualizza la stringa di caratteri ASCII che termina con un codice 00 e il cui primo byte è puntato dalla coppia Y (byte alto), A (byte basso). Questo metodo di vettorizzazione è d'altronde molto usato dal sistema operativo.
La sequenza di chiamata è quindi:

```
LDA #<INDIRIZZO
LDY #>INDIRIZZO
JSR STROUT
```

È preceduta da un posizionamento del cursore come sopra oppure si visualizza a partire dalla posizione in cui si è.

Esercizio 5.10 *Scrivete "BYE BYE" al centro dello schermo.*

Le routine annesse che si può essere portati ad utilizzare sono:

CRLF : (\$AAD7) Scrive un 'Return'.
CLR : (\$E544) Vuota lo schermo.
SCROLL : (\$E8EA) Fa scorrere lo schermo di una linea verso il basso (per far scorrere verso l'alto è \$E965).

Queste tre ultime routine si chiamano con un semplice JSR. Non ci sono sequenze di chiamata.
Le routine di input sono assai semplici:

GET : (\$FFE4) Prende un carattere al volo (ritorna il suo codice nell'accumulatore). Mette 0 se non si è avuto nessun carattere.

RDT : ($FFE4) Attende che si prema un tasto quando lo si è premuto, ritorna il codice in accumulatore.

INPUT : ($A460) Simula l'INPUT del Basic: si accetta una stringa di caratteri terminata da RETURN e la si invia nel buffer di ingresso del Basic ($200-$258). Il cursore lampeggia e c'è un punto interrogativo. L'unica differenza dal Basic è che le virgole e i due punti non sono accettati.

Una routine utile è la **CHRGET** che prende, in accumulatore, un carattere dal testo del Basic. CHRGET è in RAM in $0073. Il carattere da prendere è puntato da $7A e $7B. Un altro punto d'ingresso è $79 dove, poiché il puntatore non è stato incrementato, si riprende il carattere che si era già preso. La routine salta gli spazi e si arresta ai due punti.

Routine aritmetiche

Le routine aritmetiche utilizzano due accumulatori **ACC1** ($61-$66) e **ACC2** ($69-$6E) col formato descritto al capitolo 2 per contenere i numeri reali.
Come il 6502 possiede le istruzioni LDA e STA, noi disporremo delle routine per caricare e memorizzare questi accumulatori.
Le routine sono:

MACC1 : ($BBA2) che trasferisce in ACC1 un numero nel formato virgola mobile puntato da A (basso) e Y (alto). Si effettua contemporaneamente la conversione dal formato in virgola mobile ordinario al formato accumulatore.

MACC2 : ($BA8C) che svolge lo stesso compito per ACC2.

ACC1M : ($BBD4) che trasferisce ACC1 verso il gruppo di 5 byte puntati da X (basso) e Y (alto), con conversione dal formato accumulatore al formato in virgola mobile ordinario.

Non c'è ACC2M perché il risultato di un'operazione si trova sempre in ACC1.

ACC21 : ($BBFC) che trasferisce ACC2 in ACC1.
ACC12 : ($BC0C) che trasferisce ACC1 in ACC2.

Alcune routine effettuano un'operazione sull'accumulatore:

ZERO : ($B8F7) mette a zero ACC1.
OPPOS : ($B947) trasforma ACC1 nel suo opposto.
ARRON : ($BC1B) arrotonda ACC1.
NORMAL : ($B8D7) normalizza ACC1 (rimette nella forma 0,...).

Ricordiamo che al momento della chiamata del vostro sottoprogramma in linguaggio macchina con Z=USR(**K**), il valore di **K** passa in ACC1 e che il valore lasciato in ACC1 al momento del ritorno si ritrova in **Z**. Le routine aritmetiche sono nella forma ACC1 ← ACC1 op M dove M identifica il numero puntato da A (basso) e Y (alto). In questo caso, il nome della routine termina con una M. Le principali routine sono:

Nome	Indirizzo	Operazione
ADD	B86A	ACC1 ← ACC1 + ACC2
ADDM	B867	ACC1 ← ACC1 + (Y, A)
SUB	B853	ACC1 ← ACC1 − ACC2
SUBM	B850	ACC1 ← ACC1 − (Y, A)
MULT	BA2B	ACC1 ← ACC1 * ACC2
MULTM	BA28	ACC1 ← ACC1 * (Y, A)
DIV	BB12	ACC1 ← ACC2/ACC1
DIVM	BB0F	ACC1 ← (Y, A)/ACC1
FPEXP	BF7B	ACC1 ← ACC2 ↑ ACC1

Le funzioni aritmetiche di biblioteca effettuano l'operazione ACC1 ← f (ACC1). Ecco gli indirizzi della biblioteca del Basic.

Funzione	Indirizzo
ABS	BC58
ATN	E30E
COS	E264
EXP	BFED
INT	BCCC
LOG	B9EA
SIN	E26B
SQR	BF71
TAN	E2B4

Routine di conversione

INTFLP : ($B391) converte in ACC1 il numero intero il cui byte alto è in A e il byte basso in Y.

FLPINT : ($B1BF) converte il numero in virgola mobile che è in ACC1 in intero. Il risultato viene messo in ACC1: byte in alto $64 e byte basso $65.

FLPASC : ($BDDD) converte in ASCII il numero contenuto in ACC1. Il risultato sarà a partire da $0100 e termina con un byte 00, cioè subito pronto a essere visualizzato da STROUT se si caricano A e Y per puntare verso 0100: LDA #$00 e LDY #$01.

Conversione di una stringa in numero

Intero : fare puntare \$7A-\$7B verso la stringa (terminata da uno spazio), poi:

```
JSR CHRGET ; $0073
JSR ASCINT ; $A96B
```

Il risultato sarà in \$14 (basso) e \$15 (alto).

Reale : Fare puntare \$7A-\$7B verso la stringa (terminata da una virgola), poi:

```
JSR EXPR    ; $AD8A
```

EXPR è la routine di valutazione dell'espressione aritmetica che abbiamo già usato a pagg. 113-114. Attenzione, utilizzando anche il Basic, il puntatore (\$7A-\$7B) deve essere salvato e ripristinato.

Esempio di utilizzo

Scriveremo un sottoprogramma di prodotto fra matrici A e B 40x40. La pura versione Basic è data più avanti.
Diamo ad ogni elemento della matrice A il valore $A_{ij}=j$ per ogni i: con j che va da 0 a 40.
Ugualmente $B_{ij}=i$ per ogni j: con i che va da 0 a 40. In questo modo, ogni elemento della matrice prodotto C deve essere uguale a $C_{ij}=22140$, ciò che ci permetterà di verificare il funzionamento del programma. Potrete tuttavia notare una differenza di velocità. In 70, visualizziamo un messaggio segnalante l'inizio del prodotto e annotiamo il tempo. La durata del prodotto è visualizzata alla fine, così come un elemento della matrice risultante.

```
10 N=40:N2=20
20 DIM A(N,N),B(N,N),C(N,N)
30 FOR I=0 TO N
40 FOR J=0 TO N
50 A(I,J)=J:B(I,J)=I
60 NEXT J,I
70 PRINT"XXXX":T=TI
80 GOSUB 1000
90 PRINT C(N2,N2),TI-T
100 END
1000 FOR I=0 TO N
1010 FOR J=0 TO N
1020 C(I,J)=0
```

segue

seguito

```
1030 FOR K=0 TO N
1040 C(I,J)=C(I,J)+A(I,K)*B(K,J)
1050 NEXT K,J,I
1060 RETURN
READY.
```

Otterremo il risultato 22140 100281, il che significa che il prodotto ha preso 100281/3600=27,8 minuti!
La maggior perdita di tempo dovuta all'interprete Basic è dovuta al fatto che ad ogni iterazione del ciclo più interno (linea 1040), ogni variabile I, J, K, A, B, C è ricercata nella tabella delle variabili.
Cerchiamo di sostituire il sottoprogramma 1000 per evitare ciò. Per questo, annotiamo gli indirizzi dei primi elementi A(0,0), B(0,0) e C(0,0). È facile: l'abbiamo già fatto a pag. 132. Basta fare X=A(0,0) e in \$71-\$72, avremo l'indirizzo di A(0,0). A questo punto, l'indirizzo di A(i,j) è:
ind A(i,j) = ind A(0,0)+j*(N+1)*e+e*1
Infatti, gli elementi sono memorizzati nell'ordine:
A(0,0) A(1,0) A(2,0) .... A(N,0) A(0,1) ..... A(N,1) .....
e ogni elemento occupa e=5 byte.
Abbiamo bisogno di tre variabili di un byte: I, J, K di 6 doppi byte, puntatori verso C(0,0), A(0,0), B(0,0) e C(i,j), A(i,j), B(i,j). Utilizzeremo gli indirizzi della tabella qui sotto.

Attenzione: gli indirizzi adottati interferiscono col funzionamento della cassetta.

I	J	K	C	A	B	CIJ	AIK	BKJ
BD	BE	BF	9B,9C	9E,9F	F7,F8	F9,FA	FB,FC	FD,FE
189	190	191	155,156	158,159	247,248	249,250	251,252	253,254

Il sottoprogramma della linea 1000 diventa allora:

```
1000 X=C(0,0):POKE155,PEEK(71):POKE156,PEEK(72)
1030 X=A(0,0):POKE158,PEEK(71):POKE159,PEEK(72)
1060 X=B(0,0):POKE247,PEEK(71):POKE248,PEEK(72)
1090 SYS49152
1100 RETURN
```

Il sottoprogramma in linguaggio macchina compare qui sotto. È volontariamente che lo diamo senza commenti affinché il lettore faccia lo sforzo di comprenderlo. Si vedrà in particolare l'annidamento dei tre cicli su I, J e K.
Abbiamo, qui, messo il ciclo J all'esterno perché ciò permette una progressione più semplice dell'indirizzo C_{ij}. La progressione degli indirizzi B_{kj} e A_{ik} è più delicata.
Si noterà la sequenza di chiamata delle routine MACC1, MULTM, ACC1M e ADDM; essendo dato l'indirizzo della zona di memoria dell'operando, si caricano i registri **A** e **Y** oppure **X** e **Y** con il byte basso e il byte alto di questo indirizzo e poi basta fare JSR.
Il tempo ottenuto è 7594 sessantesimi di secondo, cioè 2,11 minuti contro 27,8. **Questo risultato, da solo, giustifica questo libro:** è necessario per talune applicazioni, programmare in linguaggio macchina!
Ripetiamo ancora che non è tanto nelle routine aritmetiche che il Basic perde del tempo (abbiamo impiegato le stesse) ma nella ricerca dell'indirizzo di una variabile, ogni volta che è referenziata.

```
00001  0000              N       =   40
00002  0000              E       =   5
00003  0000              Q       =   205
00004  0000              R       =   200
00005  0000              I       =   $BD
00006  0000              J       =   $BE
00007  0000              K       =   $BF
00008  0000              C0      =   $9B
00009  0000              A0      =   $9E
00010  0000              B0      =   $F7
00011  0000              CIJ     =   $F9
00012  0000              AIK     =   $FB
00013  0000              BKJ     =   $FD
00014  0000              ;
00015  0000              ACC1M   =   $BBD4
00016  0000              ADDM    =   $B867
00017  0000              MACC1   =   $BBA2
00018  0000              MULTM   =   $BA28
00019  0000              ZERO    =   $B8F7
00020  0000              ;
00021  0000              *       =   $C000
00022  C000              ;
00023  C000  D8          PROD    CLD
00024  C001  A5 9B               LDA C0
```

segue

seguito

```
00025  C003  85 F9              STA CIJ
00026  C005  A5 9C              LDA C0+1
00027  C007  85 FA              STA CIJ+1
00028  C009  A9 00              LDA #0
00029  C00B  85 BE              STA J
00030  C00D  A9 00      CICLOJ  LDA #0
00031  C00F  85 BD              STA I
00032  C011  20 F7 B8   CICLOI  JSR ZERO
00033  C014  A6 F9              LDX CIJ
00034  C016  A4 FA              LDY CIJ+1
00035  C018  20 D4 BB           JSR ACC1M
00036  C01B  A9 00              LDA #0
00037  C01D  85 BF              STA K
00038  C01F  A5 F7              LDA B0
00039  C021  85 FD              STA BKJ
00040  C023  A5 F8              LDA B0+1
00041  C025  85 FE              STA BKJ+1
00042  C027  A5 9E              LDA A0
00043  C029  85 FB              STA AIK
00044  C02B  A5 9F              LDA A0+1
00045  C02D  85 FC              STA AIK+1
00046  C02F  A5 FB      CICLOK  LDA AIK
00047  C031  A4 FC              LDY AIK+1
00048  C033  20 A2 BB           JSR MACC1
00049  C036  A5 FD              LDA BKJ
00050  C038  A4 FE              LDY BKJ+1
00051  C03A  20 28 BA           JSR MULTM
00052  C03D  A6 F9              LDX CIJ
00053  C03F  A4 FA              LDY CIJ+1
00054  C041  20 D4 BB           JSR ACC1M
00055  C044  A6 BF              LDX K
00056  C046  E0 28              CPX #N
00057  C048  F0 1C              BEQ INI
00058  C04A  E6 BF              INC K
00059  C04C  A5 FB              LDA AIK
00060  C04E  18                 CLC
00061  C04F  69 CD              ADC #<Q
00062  C051  85 FB              STA AIK
00063  C053  A5 FC              LDA AIK+1
00064  C055  69 00              ADC #>Q
00065  C057  85 FC              STA AIK+1
00066  C059  A5 FD              LDA BKJ
```

segue

seguito

```
00067   C05B   18                     CLC
00068   C05C   69 05                  ADC #E
00069   C05E   85 FD                  STA BKJ
00070   C060   90 CD                  BCC CICLOK
00071   C062   E6 FE                  INC BKJ+1
00072   C064   D0 C9                  BNE CICLOK
00073   C066   A6 BD         INI      LDX I
00074   C068   E0 28                  CPX #N
00075   C06A   F0 13                  BEQ INJ
00076   C06C   E6 BD                  INC I
00077   C06E   20 A7 C0               JSR INIJ
00078   C071   A5 9E                  LDA A0
00079   C073   18                     CLC
00080   C074   69 05                  ADC #E
00081   C076   85 9E                  STA A0
00082   C078   90 02                  BCC SUITE
00083   C07A   E6 9F                  INC A0+1
00084   C07C   4C 11 C0      SUITE    JMP CICLOI
00085   C07F   A6 BE         INJ      LDX J
00086   C081   E0 28                  CPX #N
00087   C083   F0 2D                  BEQ RET
00088   C085   E6 BE                  INC J
00089   C087   20 A7 C0               JSR INIJ
00090   C08A   A5 F7                  LDA B0
00091   C08C   18                     CLC
00092   C08D   69 CD                  ADC #<Q
00093   C08F   85 F7                  STA B0
00094   C091   A5 F8                  LDA B0+1
00095   C093   69 00                  ADC #>Q
00096   C095   85 F8                  STA B0+1
00097   C097   A5 9E                  LDA A0
00098   C099   38                     SEC
00099   C09A   E9 C8                  SBC #<R
00100   C09C   85 9E                  STA A0
00101   C09E   A5 9F                  LDA A0+1
00102   C0A0   E9 00                  SBC #>R
00103   C0A2   85 9F                  STA A0+1
00104   C0A4   4C 0D C0               JMP CICLOJ
00105   C0A7   A5 F9         INIJ     LDA CIJ
00106   C0A9   18                     CLC
00107   C0AA   69 05                  ADC #E
00108   C0AC   85 F9                  STA CIJ
```

segue

seguito

```
00109  C0AE  90 02              BCC RET
00110  C0B0  E6 FA              INC CIJ+1
00111  C0B2  60          RET    RTS
```

Esercizio 5.11 *Si conoscono gli indirizzi delle variabili A, B, C, e D (rispettivamente INDA, INDB, INDC e INDD). Fare il calcolo D=A*B+C.*

Routine di manipolazione dei file

I punti di entrata più interessanti sono vettorizzati (=indirizzo indipendente dalla macchina) alla fine della memoria. Sono:

FF81 : inizializzazione schermo e tastiera
FF84 : inizializzazione periferiche
FF87 : inizializzazione puntatori alla memoria
FF8A : ripristina i vettori di I/O
FF8D : setta i vettori di I/O
FF90 : comanda i messaggi di sistema operativo
FF93 : invia l'indirizzo secondario dopo "ascolta"
FF96 : invia l'indirizzo secondario dopo "parla"
FF99 : legge/setta parte alta della memoria
FF9C : legge/setta parte bassa della memoria
FF9F : scandisce la tastiera
FFA2 : setta il timeout per il bus IEEE seriale
FFA5 : riceve un byte dal bus IEEE seriale
FFA8 : invia un byte sul bus IEEE seriale
FFAB : invia "fine di parlare" sul IEEE
FFAE : invia "fine di ascolto" sul IEEE
FFB1 : invia "ascolto" sul IEEE
FFB4 : invia "parla" sul IEEE
FFB7 : legge ST (parola di stato dell'I/O)
FFBA : setta i parametri di un file
FFBD : setta il nome di un file
FFC0 : apertura file (OPEN)
FFC3 : chiusura file (CLOSE)
FFC6 : abilita periferica in ingresso
FFC9 : abilita periferica in uscita
FFCC : rimette le periferiche di default (schermo-tastiera)
FFCF : input di un carattere
FFD2 : output di un carattere
FFD5 : effettua LOAD
FFD8 : effettua SAVE
FFDB : abilita orologio in tempo reale
FFDE : legge orologio

FFE1 : testa il tasto STOP
FFE4 : prende un carattere (GET)
FFE7 : chiude ogni input/output
FFEA : aggiorna l'orologio in tempo reale
FFED : legge l'organizzazione dello schermo
FFF0 : legge/setta la posizione del cursore
FFF3 : legge l'indirizzo di base dell'I/O

Gli indirizzi delle variabili strategiche sono:

$99,$9A : contiene il numero di periferica dell'I/O standard del sistema (0=tastiera, 3=schermo).
$98 : numero di file aperti (deve essere < 10).
$259 : inizio della tabella dei numeri logici dei file.
$263 : inizio della tabella dei numeri di periferica.
$26D : inizio della tabella degli indirizzi secondari.
$90 : variabile di stato ST (0=R.A.S.; $40=fine file).

Per il file corrente, il sistema operativo fornisce gli indirizzi seguenti:

$B7 : numero di caratteri nel nome del file.
$B8 : numero logico.
$B9 : indirizzo secondario.
$BA : numero periferica.
$BB-$BC : puntatore alla stringa=nome del file.

La routine FFC0 è tale che, per fare un'apertura, si utilizza la sequenza:

```
$B8      ←  n. logico
$B8      ←  periferica
$B8      ←  indirizzo secondario
$B8      ←  numero caratteri
$B8      ←  indirizzo del nome
JSR $FFC0
```

Esercizio 5.12 *Inizializzare il disco 0 sull'unità 8.*

La routine FFC3 (CLOSE) è chiamata dopo aver messo il numero logico nell'accumulatore A.
Si può effettuare un caricamento di programma grazie alla sequenza:

```
$B7      ←  lunghezza del nome
$BA      ←  periferica
$BB,$BC  ←  puntatore al nome
JSR $FFD5
```

Infine, un'apertura semplificata può essere effettuata dalla serie seguente che suppone che il numero logico, il numero di periferica, l'indirizzo secondario siano stati messi in testa alle loro tabelle rispettive e che il numero di file aperti sia stato messo a 1:

input : X ← n. logico
JSR $FFC6

output : X ← n. logico
JSR $FFC9

Gli input/output carattere per carattere si faranno allora con:

input : JSR $FFCF (il carattere letto va in A)

output : A ← carattere
JSR $FFD2

La routine $FFFC chiude tutti i file. Si può terminare il programma con JMP o JSR $A474 che è il ritorno al READY del Basic.

Esempio di utilizzo

Fate il tentativo seguente:

1. Inserite il programma seguente col monitor in linguaggio macchina:

```
C000 20 C0 FF       JSR $FFFC ; OPEN
C003 A2 04          LDX #4
C005 20 C9 FF       JSR $FFC9 ; setta l'unità 4 per l'output
C008 A9 41       LP LDA #$41  ; codice di A
C00A 20 D2 FF       JSR $FFD2 ; output
C00D AD 8D 02       LDA $28D  ; tasto SHIFT
C010 F0 F6          BEQ LP    ; ricicla se SHIFT non è premuto
C012 A9 0D          LDA #$0D  ; return
C014 20 D2 FF       JSR $FFD2
C017 20 C3 FF       JSR $FFC3 ; CLOSE sapendo che D2 contiene 4
C01A 20 CC FF       JSR $FFCC ; riabilita lo schermo
C01D 00             BRK
```

2. Col monitor, inserite i seguenti valori:

$B7 : 00 lunghezza nome 0.
$B8 : 04 numero logico di file 4.
$B9 : 00 indirizzo secondario 0.
$BA : 04 numero di periferica 4 stampante.

3. Fate: .G C000

Sulla stampante verranno stampate delle A finché non premete il tasto SHIFT.

CONCLUSIONE

Eccoci arrivati alla fine di questo libro.
Ci ha mostrato che la programmazione in linguaggio macchina non è poi così difficile, a patto di utilizzare l'assembler simbolico. Come tutte le programmazioni, richiede attenzione, ecco tutto.
Ci ha anche mostrato che la programmazione in linguaggio macchina ha dei vantaggi che la rendono indispensabile in alcune applicazioni: lo si è visto a proposito del disegno animato e del prodotto di matrici in cui il fattore velocità è preponderante.
Non abbiamo potuto, in questo libro, andare così lontano come avremmo voluto nella descrizione delle routine di sistema operativo.
D'altra parte, abbiamo completamente tralasciato la programmazione dell'input/output, delle interruzioni, ecc.
Il fatto è che, oltre la programmazione del 6510, che era il soggetto di questo libro, gli input/output mettono in gioco la programmazione delle interfacce annesse (CIA e VIC) che accompagnano il 6510 per formare il C64.

Appendice

Appendice 1

Set di istruzioni del 6502/6510

ABBREVIAZIONI

A	*Accumulatore*
M	*Cella di memoria*
OPER	*Operando*
P	*Registro di stato*
PC	*Program counter*
PCH	*Byte alto del PC*
PCL	*Byte basso del PC*
S	*Stack pointer*
#	*Indirizzamento immediato*
$\vee$	*OR logico*
$\wedge$	*AND logico*
$\oplus$	*OR esclusivo*
–	*Il flag non cambia*
$\surd$	*Il flag può cambiare*
*	*Aggiungere 1 se si supera il limite di pagina*
●	*Aggiungere 1 se c'è salto*

Flag

B	Break
C	Riporto
D	Modo decimale
I	Interrupt
N	Segno
V	Overfow
Z	Zero

ADC

ADC
(ADd with Carry)

Si aggiunge all'accumulatore la memoria specificata e il riporto. Si opera in modo binario o decimale. In caso di risultato nullo in modo decimale, il flag Z non è corretto.

Operazione: A + M + C → A, C

N	Z	C	I	D	V
√	√	√	–	–	√

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	ADC # Oper	69	2	2
Pagina Zero	ADC Oper	65	2	3
Pagina Zero, X	ADC Oper, X	75	2	4
Assoluto	ADC Oper	6D	3	4
Assoluto, X	ADC Oper, X	7D	3	4 (*)
Assoluto, Y	ADC Oper, Y	79	3	4 (*)
(Indiretto, X)	ADC (Oper, X)	61	2	6
(Indiretto), Y	ADC (Oper), Y	71	2	5 (*)

AND

Si effettua l'AND logico bit a bit fra l'accumulatore e la memoria specificata.

Operazione: A $\wedge$ M → A

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	AND # Oper	29	2	2
Pagina Zero	AND Oper	25	2	3
Pagina Zero, X	AND Oper, X	35	2	4
Assoluto	AND Oper	2D	3	4
Assoluto, X	AND Oper, X	3D	3	4 (*)
Assoluto, Y	AND Oper, Y	39	3	4 (*)
(Indiretto, X)	AND (Oper, X)	21	2	6
(Indiretto), Y	AND (Oper), Y	31	2	5

ASL

ASL
(Arithmetic Shift Left)

Si fa scorrere a sinistra (di un bit) l'accumulatore o una memoria. Uno zero entra a destra mentre il bit uscente a sinistra va nel riporto.

Operazione: C ← | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ← 0

N Z C I D V
√ √ √ – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Accumulatore	ASL A	0A	1	2
Pagina Zero	ASL Oper	06	2	5
Pagina Zero, X	ASL Oper, X	16	2	6
Assoluto	ASL Oper	0E	3	6
Assoluto, X	ASL Oper, X	1E	3	7

BCC

BCC
(Branch on Carry Clear)

Se il flag C = 0, si salta all'indirizzo indicato, altrimenti si continua in sequenza.

N Z C I D V
– – – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Relativo	BCC Oper	90	2	2 (*) •

BCS

BCS
(Branch on Carry Set)

Se il flag C = 1, si salta all'indirizzo indicato, altrimenti si continua in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. bytes	Num. cicli
Relativo	BCS	Oper	B0	2	2 (*) •

BEQ

BEQ
(Branch on EQual)

Se il flag Z = 1 (cioè se l'ultimo risultato è 0 o se l'ultimo confronto ha dato uguaglianza), si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Relativo	BEQ	Oper	F0	2	2 (*) •

BIT

BIT
(BIt Test)

Si effettua l'AND virtuale (cioè il risultato non è messo in A, che resta immutato), fra l'accumulatore e la memoria specificata e il flag Z è settato di conseguenza. Inoltre, il bit 6 e 7 della memoria sono copiati rispettivamente in V e N.

Operazione: A $\wedge$ M, M7 $\rightarrow$ N, M6 $\rightarrow$ V

N	Z	C	I	D	V
M_7	√	–	–	–	M_6

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Pagina Zero	BIT	Oper	24	2	3
Assoluto	BIT	Oper	2C	3	4

BMI

BMI
(Branch on MInus)

Se il flag N = 1 (risultato negativo), si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Relativo	BMI	Oper	30	2	2 (*) •

BNE

BNE
(Branch if Not Equal)

Se il flag Z = 0 (cioè se l'ultimo risultato è diverso da 0 o se l'ultimo confronto non ha dato uguaglianza), si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Relativo	BNE Oper	D0	2	2 (*) •

BPL

BPL
(Branch if PLus)

Se il flag N = 0 (risultato ⩾ 0), si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Relativo	BPL Oper	10	2	2 (*)

BRK

BRK
(BReaK)

Si mette a 1 il flag B poi si simula una interruzione, vale a dire che si mettono in pila PC e P poi si salta all'indirizzo contenuto nel vettore di interruzione (FFFE).

N.B. - il valore di PC messo in pila è l'indirizzo di BRK+2 come se BRK occupasse 2 byte.
- il flag B è messo a 1 per distinguere da una interruzione IRQ che ha lo stesso vettore.

N	Z	C	I	D	V
–	–	–	1	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	BRK	00	1	7

BVC

BVC
(Branch on oVerflow Clear)

Se il flag V = 0 si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Relativo	BVC Oper	50	2	2 (*) •

BVS

BVS
(Branch on oVerflow Set)

Se il flag V = 1 si salta all'indirizzo indicato, altrimenti si prosegue in sequenza.

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Relativo	BVS Oper	70	2	2 (*) ●

CLC

CLC
(CLear Carry)

Si forza a 0 il flag di riporto (serve in particolare prima di ADC per fare un'addizione senza riporto).

Operazione: 0 → C

N	Z	C	I	D	V
–	–	0	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	CLC	18	1	2

CLD

CLD
(CLear Decimal mode)

Si forza a 0 il flag D per mettere l'unità aritmetica in modo decimale (in vista di ADC o SBC).

Operazione: 0 → D

N	Z	C	I	D	V
–	–	–	–	0	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	CLD	D8	1	2

CLI

CLI
(CLear Interrupt inhibit flag)

Si forza a 0 il flag I di disabilitazione delle interruzioni IRQ, quindi si autorizzano tali interruzioni. Una routine di interruzione che deve lei stessa essere interrompibile deve utilizzare questa istruzione perché l'arrivo di una interruzione disabilita le interruzioni seguenti. CLI serve anche alla fine di sequenze critiche durante le quali si disabilitano le interruzioni con SEI.

Operazione: 0 → I

N	Z	C	I	D	V
–	–	–	0	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	CLI	58	1	2

CLV

CLV
(CLear oVerflow)

Si forza a 0 il flag V.
NOTA: unica istruzione del genere il cui reciproco sarebbe SEV non esiste sul 6502.

Operazione: 0 → V

N	Z	C	I	D	V
–	–	–	–	–	0

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	CLV	B8	1	2

CMP

CMP
(CoMPare accumulator)

Si effettua la sottrazione virtuale (cioè il risultato non è messo in A che rimane immutato) accumulatore – memoria e si settano i flag N, Z e C: Z è messo a 1 se c'è uguaglianza; C è messo a 1 se A $\geqslant$ M (i numeri sono considerati senza segno). Notate che è C il più determinante. Per prevedere lo stato di N fate A $+/\overline{M}$: N sarà corretto se non c'è overflow; V resta immutato.
La caratteristica più importante dell'istruzione è che A resta immutato, da cui la possibilità di confronti in serie.

Operazione: A−M

N	Z	C	I	D	V
√	√	√	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Immediato	CMP	# Oper	C9	2	2
Pagina Zero	CMP	Oper	C5	2	3
Pagina Zero, X	CMP	Oper, X	D5	2	4
Assoluto	CMP	Oper	CD	3	4
Assoluto, X	CMP	Oper, X	DD	3	4 (*)
Assoluto, Y	CMP	Oper, Y	D9	3	4 (*)
(Indiretto, X)	CMP	(Oper, X)	C1	2	6
(Indiretto), Y	CMP	(Oper), Y	D1	2	5 (*)

CPX

CPX
(ComPare X register)

Si effettua la sottrazione virtuale (cioè il risultato non è messo in X che rimane immutato) registro X – memoria e si settano i flag N, Z e C: Z è messo a 1 se c'è uguaglianza; C è messo a 1 se X ⩾ M (i numeri sono considerati senza segno). Notate che è C il più determinante. Per prevedere lo stato di N fate $X +/\overline{M}$: N sarà corretto se non c'è overflow; V resta immutato.
La caratteristica più importante dell'istruzione è che X resta immutato, da cui la possibilità di confronti in serie.

Operazione: X−M

N	Z	C	I	D	V
√	√	√	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	CPX # Oper	E0	2	2
Pagina Zero	CPX Oper	E4	2	3
Assoluto	CPX Oper	EC	3	4

CPY

CPY
(ComPare Y register)

Si effettua la sottrazione virtuale (cioè il risultato non è messo in Y che rimane immutato) registro Y – memoria e si settano i flag N, Z e C: Z è messo a 1 se c'è uguaglianza; C è messo a 1 se Y $\geq$ M (i numeri sono considerati senza segno). Notate che è C il più determinante. Per prevedere lo stato di N fate Y $+/\overline{M}$: N sarà corretto se non c'è overflow; V resta immutato.
La caratteristica più importante dell'istruzione è che Y resta immutato, da cui la possibilità di confronti in serie.

Operazione: Y−M

N	Z	C	I	D	V
√	√	√	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	CPY # Oper	C0	2	2
Pagina Zero	CPY Oper	C4	2	3
Assoluto	CPY Oper	CC	3	4

DEC

DEC
(DECrement memory)

Si decrementa di 1 il contenuto della memoria indicata.

Operazione: M − 1 → M

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Pagina Zero	DEC Oper	C6	2	5
Pagina Zero, X	DEC Oper, X	D6	2	6
Assoluto	DEC Oper	CE	3	6
Assoluto, X	DEC Oper, X	DE	3	7

DEX

DEX
(DEcrement X register)

Si decrementa di 1 il contenuto del registro indice X.

Operazione: X − 1 → X

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	DEX	CA	1	2

DEY

DEY
(DEcrement Y register)

Si decrementa di 1 il contenuto del registro indice Y.

Operazione: Y − 1 → Y

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	DEY	88	1	2

EOR

EOR
(Exclusive OR)

Si effettua l'OR esclusivo fra l'accumulatore e la memoria indicata.

Operazione: A ⊕ M → A

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	EOR # Oper	49	2	2
Pagina Zero	EOR Oper	45	2	3
Pagina Zero, X	EOR Oper, X	55	2	4
Assoluto	EOR Oper	4D	3	4
Assoluto, X	EOR Oper, X	5D	3	4 (*)
Assoluto, Y	EOR Oper, Y	59	3	4 (*)
(Indiretto, X)	EOR (Oper, X)	41	2	6
(Indiretto), Y	EOR (Oper), Y	51	2	5 (*)

INC

INC
(INCrement memory)

Si incrementa di 1 il contenuto della memoria indicata.

Operazione: M + 1 → M

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Pagina Zero	INC Oper	E6	2	5
Pagina Zero, X	INC Oper, X	F6	2	6
Assoluto	INC Oper	EE	3	6
Assoluto, X	INC Oper, X	FE	3	7

INX

INX
(INcrement X register)

Si incrementa di 1 il contenuto del registro indice X.

Operazione: X + 1 → X

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	INX	E8	1	2

INY

INY
(INcrement Y register)

Si incrementa di 1 il contenuto del registro indice Y.

Operazione: Y + 1 → Y

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	INY	C8	1	2

JMP

JMP
(JuMP)

Si salta all'indirizzo indicato.

Operazione: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Assoluto	JMP Oper	4C	3	3
Indiretto	JMP (Oper)	6C	3	5

JSR

JSR
(Jump to SubRoutine)

Si salva PC (1) nella pila per costituire l'indirizzo di ritorno poi si salta all'indirizzo indicato.

Operazione: PC+2 ↓ , (PC+1) → PCL
(PC+2) → PCH

N Z C I D V
– – – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Assoluto	JSR Oper	20	3	6

(1) È l'indirizzo di JSR + 2 = indirizzo istruzione seguente − 1 che è salvato perché RTS incrementa PC.

LDA

LDA
(LoaD Accumulator)

Si mette, in accumulatore, il contenuto della memoria indicata (la memoria non viene alterata).

Operazione: M → A

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Immediato	LDA	# Oper	A9	2	2
Pagina Zero	LDA	Oper	A5	2	3
Pagina Zero, X	LDA	Oper, X	B5	2	4
Assoluto	LDA	Oper	AD	3	4
Assoluto, X	LDA	Oper, X	BD	3	4 (*)
Assoluto, Y	LDA	Oper, Y	B9	3	4 (*)
(Indiretto, X)	LDA	(Oper, X)	A1	2	6
(Indiretto), Y	LDA	(Oper), Y	B1	2	5 (*)

LDX

LDX
(LoaD X register)

Si mette, nel registro X, il contenuto della memoria indicata (la memoria non viene alterata).

Operazione: M → X

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Immediato	LDX	# Oper	A2	2	2
Pagina Zero	LDX	Oper	A6	2	3
Pagina Zero, Y	LDX	Oper, Y	B6	2	4
Assoluto	LDX	Oper	AE	3	4
Assoluto, Y	LDX	Oper, Y	BE	3	4 (*)

LDY

LDY
(LoaD X register)

Si mette, nel registro Y, il contenuto della memoria indicata (la memoria non viene alterata).

Operazione: M → Y

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	LDY # Oper	A0	2	2
Pagina Zero	LDY Oper	A4	2	3
Pagina Zero, X	LDY Oper, X	B4	2	4
Assoluto	LDY Oper	AC	3	4
Assoluto, X	LDY Oper, X	BC	3	4 (*)

LSR

LSR
(Logical Shift right)

Si fa scorrere a destra (di un bit) l'accumulatore o una memoria. Uno zero entra a sinistra mentre il bit uscente a destra va nel riporto.

Operazione: 0 → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | → C

N	Z	C	I	D	V
0	√	√	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Accumulatore	LSR A	4A	1	2
Pagina Zero	LSR Oper	46	2	5
Pagina Zero, X	LSR Oper, X	56	2	6
Assoluto	LSR Oper	4E	3	6
Assoluto, X	LSR Oper, X	5E	3	7

NOP

NOP
(No OPeration)

Istruzione muta: non viene svolta alcuna azione. La durata è di due cicli. È usata sia per rimpiazzare istruzioni soppresse durante la correzione del programma sia per allungare i cicli di ritardo.

N Z C I D V
– – – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	NOP	EA	1	2

ORA

ORA
(OR Accumulator)

Si effettua l'OR bit a bit fra l'accumulatore e la memoria indicata.

Operazione: A ∨ M → A

N Z C I D V
√ √ – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	ORA # Oper	09	2	2
Pagina Zero	ORA Oper	05	2	3
Pagina Zero, X	ORA Oper, X	15	2	4
Assoluto	ORA Oper	0D	3	4
Assoluto, X	ORA Oper, X	1D	3	4 (*)
Assoluto, Y	ORA Oper, Y	19	3	4 (*)
(Indiretto, X)	ORA (Oper, X)	01	2	6
(Indiretto), Y	ORA (Oper), Y	11	2	5

PHA

PHA
(PusH Accumulator)

Si mette il contenuto dell'accumulatore alla sommità della pila e si aggiorna il puntatore alla pila. A resta intatto.

Operazione: A ↓

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	PHA	48	1	3

PHP

PHP
(PusH Processor Status word)

Si mette il contenuto del registro di stato alla sommità della pila e si aggiorna il puntatore alla pila. P resta intatto.

Operazione: P ↓

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	PHP	08	1	3

PLA

PLA
(PulL Accumulator)

Si mette il contenuto della sommità della pila nell'accumulatore e si aggiorna il puntatore alla pila.

Operazione: A ↑

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	PLA	68	1	4

PLP

PLP
(PulL Processor status word)

Si mette il contenuto della sommità della pila nel registro di stato e si aggiorna il puntatore alla pila.

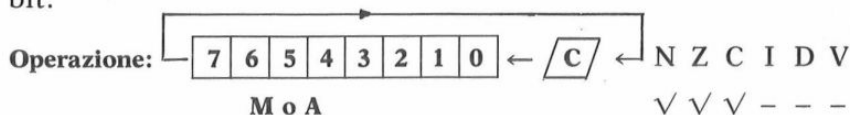
Operazione: P ↑

N	Z	C	I	D	V
√	√	√	√	√	√

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	PLP	8	1	4

ROL

ROL
(ROtate Left)

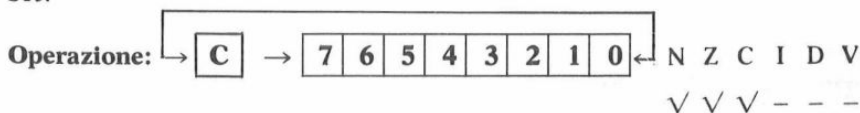
Si fa ruotare a sinistra (di un bit) l'accumulatore o una memoria. Il vecchio valore del bit di riporto entra a destra mentre il bit che esce a sinistra costituisce il nuovo valore di C. Si tratta di una rotazione su 9 bit.

Operazione: ⌐ 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 ← C ← ┘ N Z C I D V

M o A √ √ √ – – –

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Accumulatore	ROL	A	2A	1	2
Pagina Zero	ROL	Oper	26	2	5
Pagina Zero, X	ROL	Oper, X	36	2	6
Assoluto	ROL	Oper	2E	3	6
Assoluto, X	ROL	Oper, X	3E	3	7

ROR

ROR
(ROtate Right)

Si fa ruotare a destra (di un bit) l'accumulatore o una memoria. Il vecchio valore del bit di riporto entra a sinistra mentre il bit che esce a destra costituisce il nuovo valore di C. Si tratta di una rotazione su 9 bit.

Operazione: → C → 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 ← N Z C I D V

√ √ √ – – –

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Accumulatore	ROR	A	6A	1	2
Pagina Zero	ROR	Oper	66	2	5
Pagina Zero, X	ROR	Oper, X	76	2	6
Assoluto	ROR	Oper	6E	3	6
Assoluto, X	ROR	Oper, X	7E	3	7

RTI

RTI
(ReTurn from Interrupt)

Ritorno dalla routine di interruzione: si recupera dalla pila PC e P che vi erano stati salvati dal meccanismo di interruzione e si aggiorna il puntatore alla pila. Si riprende l'esecuzione da dove si era al momento dell'interruzione.

Operazione: P ↑ PC ↑

N	Z	C	I	D	V
√	√	√	√	√	√

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	RTI	40	1	6

RTS

RTS
(ReTurn from Subroutine)

Si recupera dalla pila PC che era stato salvato dall'ultimo JSR. Si riprende l'esecuzione dall'istruzione successiva alla chiamata al sottoprogramma.

Operazione: PC ↑, PC + 1 → PC

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	RTS	60	1	6

SBC

SBC
(SuBctratx with Carry)

Si sottrae dall'accumulatore il contenuto della memoria indicata e anche l'opposto del riporto (cioè il prestito). Si opera in modo decimale o binario. In caso di risultato in modo decimale, il flag Z non è corretto.

Operazione: A − M − $\bar{C}$ → A

N	Z	C	I	D	V
√	√	√	−	−	√

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Immediato	SBC # Oper	E9	2	2
Pagina Zero	SBC Oper	E5	2	3
Pagina Zero, X	SBC Oper, X	F5	2	4
Assoluto	SBC Oper	ED	3	4
Assoluto, X	SBC Oper, X	FD	3	4 (*)
Assoluto, Y	SBC Oper, Y	F9	3	4 (*)
(Indiretto, X)	SBC (Oper, X)	E1	2	6
(Indiretto), Y	SBC (Oper), Y	F1	2	5 (*)

SEC

SEC
(SEt Carry)

Si forza a 1 il flag C (serve in particolare prima di SBC per fare una sottrazione senza riporto).

Operazione: 1 → C

N	Z	C	I	D	V
−	−	1	−	−	−

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	SEC	38	1	2

SED

SED
(SEt Decimal mode)

Si forza a 1 il flag D per mettere l'unità aritmetica in modo decimale (in vista di ADC o SBC).

Operazione: 1 → D

N	Z	C	I	D	V
–	–	–	–	1	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	SED	F8	1	2

SEI

SEI
(SEt Interrupt inhibit flag)

Si forza a 1 il flag I di disabilitazione delle interruzioni IRQ, quindi si "mascherano" queste interruzioni. Se la domanda di interruzione è mantenuta (segnale/IRQ mantenuto a 0), l'interruzione sarà presa in considerazione quando il flag sarà rimesso a 0. Questa istruzione è necessaria all'ingresso di una sequenza critica durante la quale le interruzioni devono essere disabilitate (per esempio durante i cambiamenti del vettore di interruzione).

Operazione: 1 → I

N	Z	C	I	D	V
–	–	–	1	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	SEI	78	1	2

STA

STA
(STore Accumulator)

Si mette il contenuto dell'accumulatore nella memoria indicata. A resta inalterato.

Operazione: A → M

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Pagina Zero	STA	Oper	85	2	3
Pagina Zero, X	STA	Oper, X	95	2	4
Assoluto	STA	Oper	8D	3	4
Assoluto, X	STA	Oper, X	9D	3	4 (*)
Assoluto, Y	STA	Oper	99	3	4 (*)
(Indiretto, X)	STA	(Oper, X)	81	2	6
(Indiretto), Y	STA	(Oper), Y	91	2	5 (*)

STX

STX
(STore X register)

Si mette il contenuto del registro indice X nella memoria indicata. X resta inalterato.

Operazione: X → M

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Pagina Zero	STX	Oper	86	2	3
Pagina Zero, Y	STX	Oper, Y	96	2	4
Assoluto	STX	Oper	8E	3	4

STY

STY
(STore Y register)

Si mette il contenuto del registro indice Y nella memoria indicata. Y resta inalterato.

Operazione: Y → M

N Z C I D V
\- - - - - -

Modo di indirizzamento	Forma in linguaggio assembler		Codice operativo	Num. byte	Num. cicli
Pagina Zero	STY	Oper	84	2	3
Pagina Zero, X	STY	Oper, X	94	2	4
Assoluto	STY	Oper	8C	3	4

TAX

TAX
(Transfer A to X)

Si copia il contenuto dell'accumulatore nel registro indice X. A resta inalterato.

Operazione: A → X

N Z C I D V
√ √ - - - -

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TAX	AA	1	2

TAY

TAY
(Transfer A to Y)

Si copia il contenuto dell'accumulatore nel registro indice Y. A resta inalterato.

Operazione: A → Y

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TAY	A8	1	2

TSX

TSX
(Transfer S to X)

Si copia il contenuto del puntatore della pila S nel registro indice X. S resta inalterato.
Dopo TSX, LDA $101, X legge la sommità della pila senza aggiornare S, che è uguale a PLA.

Operazione: S → X

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TSX	BA	1	2

TXA

TXA
(Transfer X to A)

Si copia il contenuto del registro indice X nell'accumulatore. X resta inalterato.

Operazione: X → A

N	Z	C	I	D	V
√	√	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TXA	8A	1	2

TXS

TXS
(Transfer X to S)

Si copia il contenuto del registro indice X nel puntatore alla pila S. X resta inalterato.
Poiché non esiste LDS, l'inizializzazione del puntatore alla pila si fa con la sequenza: LDX; TXS.

Operazione: X → S

N	Z	C	I	D	V
–	–	–	–	–	–

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TXS	9A	1	2

TYA

TYA
(Transfer Y to A)

Si copia il contenuto del registro indice Y nell'accumulatore. Y resta inalterato.

Operazione: Y → A

N Z C I D V
√ √ – – – –

Modo di indirizzamento	Forma in linguaggio assembler	Codice operativo	Num. byte	Num. cicli
Implicito	TYA	98	1	2

Appendice 2

Tabella di disassemblaggio

Questa tabella è l'inverso della seguente. In funzione del codice esadecimale AB, dà lo mnemonico e il modo di indirizzamento corrispondente. Esempio: A9 → LDA IMM (riga A col. 9). (Modo di indirizzamento assente = implicito o relativo).

A \ B	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRK	ORA IND,X				ORA PZ,X	ASL PZ,X		PHP	ORA IMM	ASL A			ORA ABS	ASL ABS	
1	BPL	ORA IND,Y				ORA PZ,X	ASL PZ,X		CLC	ORA ABS,Y				ORA ABS,X	ASL ABS,X	
2	JSR	AND IND,X			BIT PGE Z	AND PGE Z	ROL PGE Z		PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS	
3	BMI	AND IND,Y				AND PZ,X	ROL PZ,X		SEC	AND ABS,Y				AND ABS,X	ROL ABS,X	
4	RTI	EOR IND,X				EOR PGE Z	LSR PGE,Z		PHA	EOR IMM	LSR A		JMP ABS	EOR ABS	LSR ABS	
5	BVC	EOR IND,Y				EOR PZ,X	LSR PZ,X		CLI	EOR ABS,Y				EOR ABS,X	LSR ABS,X	
6	RTS	ADC IND,X				ADC PGE Z	ROR PGE Z		PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS	
7	BVS	ADC IND,Y				ADC PZ,X	ROR PZ,X		SEI	ADC ABS,Y				ADC ABS,X	ROR ABS,X	
8		STA IND,X			STY PGE 2	STA PGE Z	STX PGE Z		DEY		TXA		STY ABS	STA ABS	STX ABS	
9	BCC	STA IND,Y			STY PZ,X	STA PZ,X	STX PZ,Y		TYA	STA ABS,Y	TXS			STA ABS,X		
A	LDY IMM	LDA IND,X	LDX IMM		LDY PGE Z	LDA PGE Z	LDX PGE Z		TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS	
B	BCS	LDA IND,Y			LDY PZ,X	LDA PZ,X	LDX PZ,Y		CLV	LDA ABS,Y	TSX		LDY ABS,X	LDA ABS,X	LDX ABS,Y	
C	CPY IMM	CMP IND,X			CPY PGE Z	CMP PGE Z	DEC PGE Z		INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS	
D	BNE	CMP IND,Y				CMP PZ,X	DEC PZ,X		CLD	CMP ABS,Y				CMP ABS,X	DEC ABS,X	
E	CPX IMM	SBC IND,X			CPX PGE Z	SBC PGE Z	INC PGE Z		INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS	
F	BEQ	SBC IND,Y				SBC PZ,X	INC PZ,X		SED	SBC ABS,Y				SBC ABS,X	INC ABS,X	

Op. → Codice Operativo n → numero cicli
→ numero byte

ISTRUZIONE		IMMEDIATO			DIRETTO			PAGINA ZERO			ACCUM.			IMPLICITO			(IND,X)			(IND,Y)			PAG. Z,X			DIRETTO,X			DIRETTO,Y			RELATIVO			INDIRETTO			PAG. Z,Y			REGISTRO DI STATO	
MNEMO	EFFETTO	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V . B D I Z C	MNEMO
L D X	M → X (1)	A2	2	2	AE	4	3	A6	3	2																			BE	4	3							B6	4	2	N . . . . . Z .	L D X
L D Y	M → Y (1)	A0	2	2	AC	4	3	A4	3	2													B4	4	2	BC	4	3													N . . . . . Z .	L D Y
L S R	0 → [7 … 0] → C				4E	6	3	46	5	2	4A	2	1										56	6	2	5E	7	3													0 . . . . . Z C	L S R
N O P	NO OPERATION													EA	2	1																									. . . .	N O P
O R A	AVM → A	09	2	2	0D	4	3	05	3	2							01	6	2	11	5	2	15	4	2	1D	4	3	19	4	3										N . . . . . Z .	O R A
P H A	A → Ms S−1 → S													48	3	1																									. . . .	P H A
P H P	P → Ms S−1 → S													08	3	1																									. . . .	P H P
P L A	S+1 → S Ms → A													68	4	1																									N . . . . . Z .	P L A
P L P	S+1 → S Ms → P													28	4	1																									RIPRISTINATO	P L P
R O L	[7 … 0] ← [C] (rotazione)				2E	6	3	26	5	2	2A	2	1										36	6	2	3E	7	3													N . . . . . Z C	R O L
R O R	[C] → [7 … 0] (rotazione)				6E	6	3	66	5	2	6A	2	1										76	6	2	7E	7	3													N . . . . . Z C	R O R
R T I	RITORNO INTERRUPT													40	6	1																									RIPRISTINATO	R T I
R T S	RITORNO S/P													60	6	1																									. . . .	R T S
S B C	A − M − C → A (1) (4)	E9	2	2	ED	4	3	E5	3	2							E1	6	2	F1	5	2	F5	4	2	FD	4	3	F9	4	3										N V . . . . Z (3)	S B C
S E C	1 → C													38	2	1																									 1	S E C
S E D	1 → D													F8	2	1																									. . . . 1 . . .	S E D
S E I	1 → D													78	2	1																									. . . . . 1 . .	S E I
S T A	A → M				8D	4	3	85	3	2							81	6	2	91	6	2	95	4	2	9D	5	3	99	5	3										. . . .	S T A
S T X	X → M				8E	4	3	86	3	2																												96	4	2	. . . .	S T X
S T Y	Y → M				8C	4	3	84	3	2													94	4	2																. . . .	S T Y
T A X	1 → X													AA	2	1																									N . . . . . Z .	T A X
T A Y	A → Y													A8	2	1																									N . . . . . Z .	T A Y
T S X	S → X													BA	2	1																									N . . . . . Z .	T S X
T X A	X → A													8A	2	1																									N . . . . . Z .	T X A
T X S	X → S													9A	2	1																									. . . .	T X S
T Y A	Y → A													98	2	1																									N . . . . . Z .	T Y A

(1) Aggiungere 1 a "n" se cambiamento pagina
(2) Aggiungere 1 a "n" se salto a pagina diversa
(3) Prestito = / riporto
(4) In modo decimale il flag Z è inattivo. L'accumulatore deve essere testato per sapere se il risultato è nullo

X	Indice X
Y	Indice Y
A	Accumulatore
M	Memoria
Ms	Memoria (pila)
M_7	Memoria (bit 7)
M_6	Memoria (bit 6)

+	Addizione
−	Sottrazione
∧	And
∨	Or
⊕	Or esclusivo
n	Numero cicli
#	Numero byte

ISTRUZIONE		IMMEDIATO			DIRETTO			PAGINA ZERO			ACCUM			IMPLICITO			(IND, X)			(IND, Y)			PAG. Z,X			DIRETTO,X			DIRETTO,Y			RELATIVO			INDIRETTO			PAG. Z,Y			REGISTRO DI STATO	
MNEMO.	EFFETTO	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	7 6 5 4 3 2 1 0 N V . B D I Z C	MNEMO
A D C	A+M+C → A (1) (4)	69	2	2	6D	4	3	65	3	2							61	6	2	71	5	2	75	4	2	7D	4	3	79	4	3										N V . . . . Z C	A D C
A N D	AΛM → A (1)	29	2	2	2D	4	3	25	3	2							21	6	2	31	5	2	35	4	2	3D	4	3	39	4	3										N . . . . . Z .	A N D
A S L	C ← [7 0] ← 0				0E	6	3	06	5	2	0A	2	1										16	6	2	1E	7	3													N . . . . . Z C	A S L
B C C	SALTA SE C=0 (2)																															90	2	2							. . . .	B C C
B C S	SALTA SE C=1 (2)																															B0	2	2							. . . .	B C S
B E Q	SALTA SE Z+1 (2)																															F0	2	2							. . . .	B E Q
B I T	AΛM				2C	4	3	24	3	2																															M_7M_6 . . . . Z .	B I T
B M I	SALTA SE N=1 (2)																															30	2	2							. . . .	B M I
B N E	SALTA SE Z=0 (2)																															D0	2	2							. . . .	B N E
B P L	SALTA SE N=0 (2)																															10	2	2							. . . .	B P L
B R K	BREAK													00	7	1																									. . . 1 . 1 . .	B R K
B V C	SALTA SE V=0 (2)																															50	2	2							. . . .	B V C
B V S	SALTA SE V=1 (2)																															70	2	2							. . . .	B V S
C L C	0 → C													18	2	1																									 0	C L C
C L D	0 → D													D8	2	1																									. . . . 0 . . .	C L D
C L I	0 → I													58	2	1																									. . . . . 0 . .	C L I
C L V	0 → V													B8	2	1																									. 0	C L V
C M P	A - M	C9	2	2	CD	4	3	C5	3	2							C1	6	2	D1	5	2	D5	4	2	DD	4	3	D9	4	3										N . . . . . Z C	C M P
C P X	X - M	E0	2	2	EC	4	3	E4	3	2																															N . . . . . Z C	C P X
C P Y	Y - M	C0	2	2	CC	4	3	C4	3	2																															N . . . . . Z C	C P Y
D E C	M - 1 → M				CE	6	3	C6	5	2													D6	6	2	DE	7	3													N . . . . . Z .	D E C
D E X	X - 1 → M													CA	2	1																									N . . . . . Z .	D E X
D E Y	Y - 1 → Y													88	2	1																									N . . . . . Z .	D E Y
E O R	A ⊕ M → A (1)	49	2	2	4D	4	3	45	3	2							41	6	2	51	5	2	55	4	2	5D	4	3	59	4	3										N . . . . . Z .	E O R
I N C	M+1 M				EE	6	3	E6	5	2													F6	6	2	FE	7	3													N . . . . . Z .	I N C
I N X	X+1 → X													E8	2	1																									N . . . . . Z .	I N X
I N Y	Y+1 → Y													C8	2	1																									N . . . . . Z .	I N Y
J M P	SALTO INCOND.				4C	3	3																												6C	5	3				. . . .	J M P
J S R	CHIAMATA S/P				20	6	3																																		. . . .	J S R
L D A	M → A (1)	A9	2	2	AD	4	3	A5	3	2							A1	6	2	B1	5	2	B5	4	2	BD	4	3	B9	4	3										N . . . . . Z .	L D A

Appendice 3

Mappa della memoria del C64

(**) = indirizzo di pagina zero utilizzabile.
(*) = indirizzo di pagina zero *talvolta* utilizzabile.

Indirizzo esadecimale	Locazione decimale	Descrizione
0000	0	Registro direzione dati del circuito 6510
0001	1	Registro a 8 bit di Input/Output del circuito 6510
0002	2	Non usato (**)
0003-0004	3-4	Vettore salti: Conversione reale-intero
0005-0006	5-6	Vettore salti: Conversione intero-reale
0007	7	Carattere di ricerca
0008	8	Indicatore: Cerca le virgolette alla fine di una stringa
0009	9	Colonna di schermo dopo l'ultima TAB
000A	10	Indicatore: 0 = Carica, 1 = Verifica
000B	11	Puntatore buffer di input/numero indici
000C	12	Indicatore: Dimensione di default di un array
000D	13	Tipo di dato: $FF = Stringa, $00 = Numerico
000E	14	Tipo di dato: $80 = Intero, $00 = Reale
000F	15	Scansione istruzione DATA/Virgolette istruzione LIST/"Garbage Collection"
0010	16	Indicatore: Riferimento indice/Chiamata di funzione Utente
0011	17	Indicatore: $00 = INPUT, $40 = GET, $98 = READ
0012	18	Indicatore: Simbolo TAN/Risultato di un confronto

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
0013	19	Indicatore: Richiesta di INPUT
0014-0015	20-21	Transiente: Valore intero
0016	22	Puntatore: Stack stringhe transienti
0017-0018	23-24	Ultimo indirizzo stringhe transienti
0019-0021	25-33	Stack stringhe transienti
0022-0025	34-37	Area puntatori programmi di utilità
0026-002A	38-42	Prodotto di moltiplicazione reale (*)
002B-002C	43-44	Puntatore: Inizio del testo BASIC
002D-002E	45-46	Puntatore: Inizio variabili del BASIC
002F-0030	47-48	Puntatore: Inizio array del BASIC
0031-0032	49-50	Puntatore: Fine array del BASIC (+1)
0033-0034	51-52	Puntatore: Base della memoria stringa
0035-0036	53-54	Puntatore stringa programmi di utilità
0037-0038	55-56	Puntatore: Indirizzo più alto usato dal BASIC
0039-003A	57-58	Numero di linea corrente del BASIC
003B-003C	59-60	Numero di linea precedente del BASIC
003D-003E	61-62	Puntatore: Istruzione BASIC per CONT
003F-0040	63-64	Numero di linea DATA corrente
0041-0042	65-66	Puntatore: Indirizzo elemento corrente dell'istruzione DATA
0043-0044	67-68	Vettore: Routine di INPUT
0045-0046	68-69	Nome variabile corrente del BASIC
0047-0048	70-71	Puntatore: Dato variabile corrente del BASIC
0049-004A	73-74	Puntatore: Variabile indice per il ciclo FOR... NEXT
004B-0060	75-96	Area puntatore/dati transiente (*)
0061	97	Accumulatore reale 1: Esponente (*)
0062-0065	98-101	Accumulatore reale 1: Mantissa (*)
0066	102	Accumulatore reale 1: Segno (*)
0067	103	Puntatore: Costante di valutazione delle serie (*)
0068	104	Accumulatore reale 1: Cifra di overflow (*)
0069	105	Accumulatore reale 2: Esponente (*)
006A-006D	106-109	Accumulatore reale 2: Mantissa (*)
006E	110	Accumulatore reale 2: Segno (*)
006F	111	Risultato di confronto del segno: (*) Accumulatore 1 contro Accumulatore 2
0070	112	Accumulatore reale 1: Byte basso (*) (arrotondamento)
0071-0072	113-114	Puntatore: Buffer cassetta (*)

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
0073-008A	115-138	Sottoprocedura: Preleva il prossimo byte del testo BASIC
0079	121	Ingresso per un nuovo prelievo dello stesso byte di testo
007A-007B	122-123	Puntatore: Byte corrente del testo BASIC
008B-008F	139-143	Valore reale del seme della funzione RND
0090	144	Parola di stato dell'I/O del KERNAL: ST
0091	145	Indicatore: Tasto STOP/Tasto RVS
0092	146	Costante di misura del tempo per nastro
0093	147	Indicatore: 0 = Carica, 1 = Verifica (*)
0094	148	Indicatore: Bus seriale – Carattere bufferizzato di output
0095	149	Carattere bufferizzato per bus seriale
0096	150	Numero di sincronismo cassetta
0097	151	Area dati transiente
0098	152	Numero file aperti/Indice della tabella dei file
0099	153	Dispositivo di input di default (0)
009A	154	Dispositivo di output (CMD) di default (3)
009B	155	Parità carattere nastro (*)
009C	156	Indicatore: Ricevuto byte da nastro (*)
009D	157	Indicatore: $80 = Modo diretto, $00 = Modo Programma
009E	158	Registro errore passo 1 del nastro (*)
009F	159	Registro errore passo 2 del nastro
00A0-00A2	160-162	Clock in tempo reale (approssimato) ad 1/60 di secondo ("Jiffy")
00A3-00A4	163-164	Area dati transiente (*)
00A5	165	Contatore a ritroso di sincronizzazione cassetta (*)
00A6	166	Puntatore: Buffer di I/O del nastro (*)
00A7	167	Bit di input dell'RS-232/Cassetta Transiente (*)
00A8	168	Contatore bit di input dell'RS-232/Cassetta transiente (*)
00A9	169	Indicatore RS-232: Controllo del bit di partenza (*)
00AA	170	Buffer del byte di input dell'RS-232/Cassetta tansiente (*)
00AB	171	Parità input dell'RS-232/Contatore corto cassetta (*)

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
00AC-00AD	172-173	Puntatore: Buffer nastro/Scorrimento schermo
00AE-00AF	174-175	Indirizzi di Fine nastro/Fine programma
00B0-00B1	176-177	Costanti di misura del tempo del nastro
00B2-00B3	178-179	Puntatore: inizio buffer del nastro
00B4	180	Contatore bit di output dell'RS-232/Cassetta transiente (*)
00B5	181	Prossimo bit dell'RS-232 da inviare/Indicatore di fine nastro (EOT) (*)
00B6	182	Buffer del byte di output dell'RS-232 (*)
00B7	183	Lunghezza del nome del file corrente
00B8	184	Numero file logico corrente
00B9	185	Indirizzo secondario corrente
00BA	186	Numero del dispositivo corrente
00BB-00BC	187-188	Puntatore: Nome del file corrente
00BD	189	Parità output dell'RS-232/Cassetta transiente (*)
00BE	190	Contatore blocco Read/Write cassetta
00BF	191	Buffer parola seriale (*)
00C0	192	Arresto motore del nastro
00C1-00C2	193-194	Indirizzo di partenza dell'I/O
00C3-00C4	195-196	Carico nastro transiente
00C5	197	Tasto corrente premuto: CHR$ (n) 0 = Nessun tasto
00C6	198	Numero caratteri nel buffer della tastiera (coda)
00C7	199	Indicatore: Stampa caratteri inversi: 1 = Si, 0 = Non usato
00C8	200	Puntatore: Fine linea logica per INPUT
00C9-00CA	201-202	Posizione (X,Y) del cursore all'inizio di INPUT
00CB	203	Indicatore: Stampa i caratteri ottenuti tenendo premuto il tasto SHIFT
00CC	204	Abilitatore del lampeggio: 0 = Lampeggio
00CD	205	Timer: Conto alla rovescia per cursore bistabile
00CE	206	Carattere sotto il cursore
00CF	207	Indicatore: Ultima impostazione cursore (lampeggio/fisso)
00D0	208	Indicatore: INPUT o GET da tastiera
00D1-00D2	209-210	Puntatore: Indirizzo della linea di schermo corrente
00D3	211	Colonna del cursore sulla linea corrente
00D4	212	Indicatore: Editor modo "quote", $00 = NO

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
00D5	213	Lunghezza linea di schermo fisica
00D6	214	Numero linea fisica attuale del cursore
00D7	215	Area dati transiente
00D8	216	Indicatore: Modo inserimento > 0 = # INST
00D9-00F2	217-242	Tavola collegamenti della linea dello schermo/ Editor transiente
00F3-00F4	243-244	Puntatore: Locazione corrente della RAM colore dello schermo
00F5-00F6	245-246	Vettore: Tavola di decodificazione della tastiera
00F7-00F8	247-248	Puntatore al buffer di input (*) dell'RS-232
00F9-00FA	249-250	Puntatore al buffer di output (*) dell'RS-232
00FB-00FE	251-254	Libera Pagina 0 per programmi Utente (**)
00FF	255	Area dati transiente del BASIC
0100-01FF	256-511	Area stack sistema del microprocessore
0100-010A	256-266	Fluttuante per area di lavoro stringa
0100-013E	256-318	Registro errori di input del nastro
0200-0258	512-600	Buffer di INPUT del sistema
0259-0262	601-610	Tabella KERNAL: Numero file logici attivi
0263-026C	611-620	Tabella KERNAL: Numero dispositivo per ogni file
026D-0276	621-630	Tabella KERNAL: Indirizzo secondario di ogni file
0277-0280	631-640	Coda del buffer della tastiera (FIFO)
0281-0282	641-642	Puntatore: Base della memoria per Sistema Operativo
0283-0284	643-644	Puntatore: Cima della memoria per Sistema Operativo
0285	645	Indicatore: Variabile KERNAL per supero Tempo dell'IEEE
0286	646	Codice colore del carattere corrente
0287	647	Colore di fondo sotto il cursore
0288	648	Cima della memoria schermo (pagina)
0289	649	Misura del buffer della tastiera
028A	650	Indicatore: Ripete il tasto battuto, $80 = Ripete
028B	651	Ripete il contatore velocità
028C	652	Ripete il contatore ritardo
028D	653	Indicatore: Tasto SHIFT della tastiera/Tasto CTRL/Tasto C=
028E	654	Ultima configurazione ottenuta con il tasto

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
		SHIFT della tastiera
028F-0290	655-656	Vettore: Preparazione tabella tastiera
0291	657	Indicatore: $00 = Disabilita tasti SHIFT, $80 = Abilita tasti SHIFT
0292	658	Indicatore: Scorrimento automatico verso il basso, 0 = ON
0293	659	RS-232: Immagine registro di controllo del 6551
0294	660	RS-232: Immagine del registro di comando del 6551
0295-0296	661-662	BPS RS-232 USA non standard (Tempo/2-100)
0297	663	RS-232: Immagine del registro di stato del 6551
0298	664	Numero di bit dell'RS-232 rimasti da inviare
0299-029A	665-666	Trasmittanza dell'RS-232: Tempo per un bit completo (nsec)
029B	667	Indice RS-232 per termine buffer input
029C	668	Inizio del buffer di input dell'RS-232 (pagina)
029D	669	Inizio del buffer di output dell'RS-232 (pagina)
029E	670	Indice RS-232 per termine buffer output
029F-02A0	671-672	Contiene il vettore IRQ durante l'I/O del nastro
02A1	673	Abilita l'RS-232
02A2	674	Lettura di TOD durante I/O cassetta
02A3	675	Memorizzazione transiente per lettura cassetta
02A4	676	Indicatore DI IRQ transiente per lettura cassetta
02A5	677	Transiente per indice di linea
02A6	678	Indicatore PAL/NTSC, 0 = NTSC, 1 = PAL
02A7-02FF	679-676	Non usati
0300-0301	768-769	Vettore: Stampa i messaggi di errore del BASIC
0302-0303	770-771	Vettore: Partenza a caldo del BASIC
0304-0305	772-773	Vettore: Testo BASIC "tokenizzato"
0306-0307	774-775	Vettore: Lista del testo BASIC
0308-0309	776-777	Vettore: Invio caratteri BASIC
030A-030B	778-779	Vettore: Valutazione "token" del BASIC
030C	780	Memorizzazione del registro A del 6502
030D	781	Memorizzazione del registro X
030E	782	Memorizzazione del registro Y
030F	783	Memorizzazione del registro SP
0310	784	Istruzione di salto della funzione USR
0311-0312	785-786	Byte basso/alto dell'indirizzo di USR

segue

seguito

Indirizzo esadecimale	Locazione decimale	Descrizione
0313	787	Non usato
0314-0315	788-789	Vettore: Interruzione hardware di IRQ
0316-0317	790-791	Vettore: Interruzione istruzione BRK
0318-0319	792-793	Vettore: Interruzione non mascherabile
031A-031B	794-795	Vettore routine OPEN del KERNAL
031C-031D	796-797	Vettore routine CLOSE del KERNAL
031E-031F	798-799	Vettore routine CHKIN del KERNAL
0320-0321	800-801	Vettore routine CHKOUT del KERNAL
0322-0323	802-803	Vettore routine CLRCHN del KERNAL
0324-0325	804-805	Vettore routine CHRIN del KERNAL
0326-0327	806-807	Vettore routine CHROUT del KERNAL
0328-0329	808-809	Vettore routine STOP del KERNAL
032A-032B	810-811	Vettore routine GETIN del KERNAL
032C-032D	812-813	Vettore routine CLALL del KERNAL
032E-032F	814-815	Vettore definito dall'Utente
0330-0331	816-817	Vettore routine LOAD del KERNAL
0332-0333	818-819	Vettore routine SAVE del KERNAL
0334-033R	820-827	Non usati
033C-03FB	828-1019	Buffer di I/O del nastro
03FC-03FF	1020-1023	Non usati
0400-07FF	1024-2047	Area memoria schermo (1024 byte)
0400-07E7	1024-2023	Matrice video (25 linee × 40 colonne)
07F8-07FF	2040-2047	Puntatori ai dati animazione
0800-9FFF	2048-40959	Spazio normale dei programmi BASIC
8000-9FFF	32768-40959	ROM cartuccia VSP (8192 byte)
A000-BFFF	40960-49151	ROM BASIC (8192 byte - 8 K RAM)
C000-CFFF	49152-53247	RAM (4096 byte)
D000-DFFF	53248-57343	Dispositivi di I/O e RAM colore, oppure ROM generatore caratteri, oppure RAM (4096 byte)
E000-EFFF	57344-65535	ROM del KERNAL (8192 byte oppure 8 K RAM)

Appendice 4

Soluzione degli esercizi

Esercizio 2.1

```
10 INPUT "HD(1) O DH(0)";X
20 IF X=0 THEN GOSUB 50000
30 IF X=1 THEN GOSUB 51000
40 GOTO 10
50000 INPUT N:H$="0123456789ABCDEF":N$=""
50010 Q=INT(N/16):R=N-16*Q:N=Q
50020 N$=MID$(H$,R+1,1)+N$
50030 IF N<>0 GOTO 50010
50040 PRINT N$:RETURN
51000 INPUT N$:L=LEN(N$):N=0
51010 FOR I=1 TO L
51020 K=ASC(MID$(N$,I,1))-48
51030 IF K>9 THEN K=K-7
51040 N=16*N+K:NEXT
51050 PRINT N:RETURN
READY.
```

50000 effettua la conversione decimale-esadecimale e 51000 fa l'inverso. Il programma principale permette di chiamarli ripetutamente.

Esercizio 2.3

No con 3 bit si possono avere $2^3 = 8$ cifre differenti. Mentre ci occorrono 10 cifre.

Esercizio 2.4

Per definizione è 33. In binario questo stesso numero sarebbe rappresentato da 21 esa.

Esercizio 2.5

Si ripristina dapprima l'ordine naturale dei byte: 4A 35. Poi si sottrae da FFFF. FFFF − 4A35 = B5CA. In seguito si aggiunge 1: B5CA + 1 = B5CB, da cui l'opposto, nell'ordine byte basso, byte alto: CB B5.

Esercizio 2.6

a) Si ha 2^{17}. 17 si scrive 11 esa da cui la caratteristica 80 + 11 = 91.
b) Basta mettere a 0 il bit di segno da cui 91 43 50 00 00.

Esercizio 2.7

L'esponente è 10 esa = 16 dec, quindi 2^{16}.

Il numero è positivo (25 = 0010 0101) e la mantissa da considerare è A5 5B 3A da cui
0,A5 5B 3A × 2^{16} = A5 5B, 3A che vale
parte intera: 10 × 4096 + 5 * 256 + 5 * 16 + 11 = 42.331
parte frazionaria: 0,3A = 0,00 11 10 10 = 1/8 + 1/16 + 1/32 + 1/128 = 0,2265625
da cui il numero 42.331,2265625
che il C64 visualizzerà nella forma 42.331,2266
Si ottiene lo stesso risultato dicendo:
A55B3A × 2^{-8} = 10.836.794/256 = 42.331,2266.

Esercizio 2.8

	classico	accumulatore
+ 5,5	83 30 00 00 00	83 B0 00 00 00 30 00
− 5,5	83 B0 00 00 00	83 B0 00 00 00 B0 FF

Esercizio 2.9

Occorrono 14/0,30103 = 46,5... bit per la mantissa.
Prenderemo quindi 6 byte da cui C MMMMMM (7 byte). Così infatti abbiamo 48 bit per la mantissa che ci assicura meglio le 14 cifre significative.

Per 16, occorrono 53,... bit cioè 7 byte da cui C MMMMMMM (8 byte in tutto).
(Si ricorda che 0,30103 = Log 2).

Esercizio 2.10

a) Ciò limita il numero dei caratteri utili a 2 (a parte $).
b) La lunghezza di una stringa è limitata a 256 caratteri.

Esercizio 2.11

Il programma termina in C00D. Poiché occorre l'indirizzo + 1, è C00E.

Esercizio 3.1

```
                                    N V X B D I Z C
Risultato = 60          SR = 71 =   0 1 1 1 0 0 0 1
```

Infatti C0 + A0 = 160 esa da cui il riporto (C = 1). Ma 60 sarebbe positivo da cui N = 0 quando si sommano due numeri negativi. C'è quindi un overflow V = 1. È perché si sommano dei numeri "troppo" negativi.

Esercizio 3.3

MAXINT = $2^{15} - 1 = 32767$
perciò − 32768 + 32767.

Esercizio 3.4

```
0000              *     = $C000
C000   A9 8A            LDA #$8A       ; PARTE BASSA
C002   18               CLC
C003   69 CD            ADC #$CD
C005   A8               TAY
C006   A9 42            LDA #$42       ; PARTE CENTRALE
C008   69 4B            ADC #$4B
C00A   AA               TAX
C00B   A9 35            LDA #$35       ; PARTE ALTA
C00D   69 27            ADC #$27
C00F   00               BRK
```

Risultato: 5C8E57.
L'istruzione TAY è la duale di TAX per il trasferimento di A in Y.

Esercizio 3.5

Vogliamo trasferire da X a Y. Il trasferimento diretto non esiste, passiamo attraverso l'accumulatore (attenzione, il suo contenuto sarà perso, guardate il rimedio: esercizio 3.26), da cui la sequenza:

TXA
TAY

Esercizio 3.6

Per C0−40, il risultato è 80 (negativo, è normale). P = 81: N = 1 (negativo), C = 1 perché nei numeri senza segno C0 > 40.

Per C0−41, il risultato è 7F (positivo perché c'è overflow). P = 71: N = 0 ma V = 1. C resta a 1 perché C0 > 41. Si vede che è il riporto a rappresentare veramente il fatto che N1 > N2.

Esercizio 3.7

```
0000                *     = $C000
C000  00            N1L   .BYTE 0      ; RISERVA
C001  00            N1H   .BYTE 0      ; MEMORIA
C002  00            N2L   .BYTE 0
C003  00            N2H   .BYTE 0
C004  AD 00 C0            LDA N1L
C007  38                  SEC
C008  ED 02 C0            SBC N2L
C00B  AA                  TAX
C00C  AD 01 C0            LDA N1H
C00F  ED 03 C0            SBC N2H
C012  00                  BRK
```

Esempio: 1836−524 dà 1312. Il riporto finale è a 1 poiché 18 > 05.

Esercizio 3.8

```
0000            * = $C000
C000  F8        SED
C001  A9 56     LDA #$56
C003  18        CLC
C004  69 41     ADC #$41
C006  AA        TAX
```

segue

seguito

```
C007  A9 19      LDA #$19
C009  69 23      ADC #$23
C00B  A8         TAY
C00C  8A         TXA
C00D  38         SEC
C00E  E9 28      SBC #$28
C010  AA         TAX
C011  98         TYA
C012  E9 10      SBC #$10
C014  D8         CLD           ; RIMETTE
C015  00         BRK           ; NORMALE
```

Se si fa G C000 si ottiene 3269; se si fa G C001, si ottiene 2C6F.

Esercizio 3.9

ASL fa in modo che C000 contenga h i j k l m n 0 mentre g passa nel riporto.
ROL fa entrare g a destra in C001 che diventa 0 a b c d e f g. Il risultato è 0 a b c d e f g h i j k l m n 0 che è il doppio di 0 0 a b c d e f g h i j k l m n.

Esercizio 3.10

```
0000             L = $22
0000             M = $23
0000             * = $C000
C000  4A         LSR A
C001  66 22      ROR L
C003  4A         LSR A
C004  66 22      ROR L
C006  4A         LSR A         ; 0U IN A
C007  66 22      ROR L         ; V0 IN L
C009  4A         LSR A
C00A  66 22      ROR L
C00C  46 22      LSR L
C00E  46 22      LSR L
C010  46 22      LSR L
C012  46 22      LSR L         ; 0V IN L
C014  0A         ASL A
C015  0A         ASL A
```

segue

seguito

```
C016   85 23        STA M       ; 0U0 IN M
C018   0A           ASL A
C019   0A           ASL A       ; U0 IN A
C01A   65 23        ADC M       ; U0+0U0
C01C   4A           LSR A       ; 10*U=1/2*
C01D   18           CLC         ; ..(U0+0U0)
C01E   65 22        ADC L       ; FINITO
C020   00           BRK
```

Per provare questo programma, caricate un valore in accumulatore con l'editor di schermo dopo un comando R. Verificate che il microprocessore non sia in modo decinale (sarebbe meglio aggiungere una CLD).

Esercizio 3.11

Basta cominciare dall'alto:

```
0000                BASSO  = $C000
0000                ALTO   = $C001
0000                *      = $C002
C002   AD 01 C0            LDA ALTO
C005   49 FF               EOR #$FF
C007   8D 01 C0            STA ALTO
C00A   AD 00 C0            LDA BASSO
C00D   49 FF               EOR #$FF
C00F   8D 00 C0            STA BASSO
C012   18                  CLC
C013   69 01               ADC #1
C015   8D 00 C0            STA BASSO
C018   AD 01 C0            LDA ALTO
C01B   69 00               ADC #0
C01D   8D 01 C0            STA ALTO
C020   00                  BRK
```

Si guadagnano una LDA e una STA, cioè 6 byte e 8 cicli macchina.

Esercizio 3.12

```
0000                L = $22
0000                M = $23
0000                * = $C000
C000   85 23        STA M        ; UV IN M   3
C002   29 0F        AND #$0F     ; 0V IN A   2
C004   85 22        STA L        ; 0V IN L   3
C006   A5 23        LDA M        ; UV IN A   3
C008   29 F0        AND #$F0     ; U0 IN A   2
C00A   85 23        STA M        ; U0 IN M   3
C00C   4A           LSR A        ;           2
C00D   4A           LSR A        ; 0U0 IN A  2
C00E   18           CLC          ;           2
C00F   65 23        ADC M        ; 0U0+U0    3
C011   4A           LSR A        ; 10*U      2
C012   18           CLC          ;           2
C013   65 22        ADC L        ;           3
C015   00           BRK
```

Il programma dell'esercizio 3.10 occupava 33 byte e si eseguiva in 71 cicli macchina, contro i 22 byte e 32 cicli macchina (abbiamo messo il numero di cicli macchina alla fine del commento). Nel programma dell'esercizio 3.10, gli scorrimenti in pagina zero prendono 5 cicli macchina e sono ripetuti a gruppi di 4.

Esercizio 3.13

```
TXA ;      setta Z se (X) = 0
BEQ LA'.
```

Esercizio 3.14

```
C000 A5 98
C002 F0 FC
C004 00           lo spiazzamento cambia!!
```

Esercizio 3.15

Installiamo in C001 il programma seguente che confronta A col contenuto di C000:

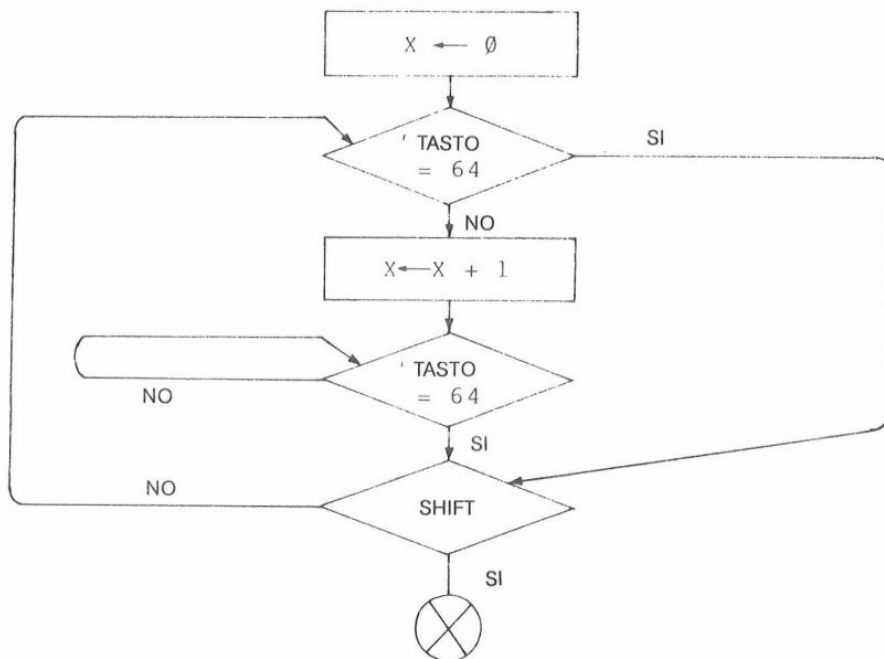
```
                          M = C000
                          * = C001
C001 CD 01 C0               CMP M
C004 00                     BRK
```

Si dà un valore a C000 col comando M e un valore ad A col comando R prima di eseguire con G C001. Basta esaminare P in seguito. A partire dalla seconda esecuzione, diamo i valori con l'editor di schermo.

Esercizio 3.16

```
        (CMP M)
        BEQ QUI
        BCS LA'
QUI                     ; caso A < = M
```

Esercizio 3.17

Flowchart:

```
0000                  TASTO = 203
0000                  SHIFT = 653
0000                  * = $C000
C000  A2 00           LDX #0
C002  A4 CB     TEST1 LDY TASTO     ; ATTENDI CHE
C004  C0 40           CPY #$40      ; VENGA PREMUTO
C006  F0 07           BEQ TEST3
C008  E8              INX
C009  A4 CB     TEST2 LDY TASTO     ; ATTENDI
C00B  C0 40           CPY #$40      ; RILASCIO
C00D  D0 FA           BNE TEST2
C00F  AD 8D 02  TEST3 LDA SHIFT
C012  F0 EE           BEQ TEST1
C014  00              BRK
```

Il tutto funziona a parte il fatto che il programma conta un tasto in più: infatti conta anche il '*Return*' che avete premuto al momento del lancio dell'esecuzione.

Esercizio 3.18

```
0000               BUFFER = $200
0000               SCHERM = $400
0000               COLORE = $D800
0000               *      = $C000
C000  13           SI     .BYTE $13
C001  09                  .BYTE $09
C002  20                  .BYTE $20
C003  0E           NO     .BYTE $0E
C004  0F                  .BYTE $0F
C005  20                  .BYTE $20
C006  A9 00               LDA #0
C008  A2 27               LDX #39
C00A  9D 00 D8     COL    STA COLORE,X
C00D  CA                  DEX
C00E  10 FA               BPL COL
C010  A6 00        TEST   LDX 0
C012  A9 3D               LDA #$3D      ; SEGNO =
C014  DD 00 02     CICLO  CMP BUFFER,X
C017  F0 13               BEQ SCRSI
```

segue

seguito

```
C019  E8                  INX
C01A  E0 59               CPX #89
C01C  90 F6               BCC CICLO
C01E  A2 00      SCRNO    LDX #0
C020  BD 03 C0   T1       LDA NO,X
C023  9D 00 04            STA SCHERM,X
C026  E8                  INX
C027  E0 03               CPX #3
C029  D0 F5               BNE T1
C02B  00                  BRK
C02C  A2 00      SCRSI    LDX #0
C02E  BD 00 C0   T2       LDA SI,X
C031  E8                  INX
C032  E0 03               CPX #3
C034  D0 F8               BNE T2
C036  00                  BRK
```

Attenzione! Non dovete essere in basso nello schermo quando fate G C006: il programma scriverà SI o NO in alto nello schermo ma il break farà scrivere tre linee che lo faranno cancellare. Il ciclo (COL) in testa assicura che ci sarà visibilità forzando "nero" nella memoria colore.

Esercizio 3.19

```
0000                   LINEA1 = $400
0000                   LINEA2 = $428
0000                          * = $C000
C000   A2 28                  LDX #40
C002   A0 00                  LDY #0
C004   BD FF 03    CICLO      LDA LINEA1-1,X
C007   99 28 04               STA LINEA2,Y
C00A   C8                     INY
C00B   CA                     DEX
C00C   D0 F6                  BNE CICLO
C00E   00                     BRK
```

Nota. Scrivete lo stesso programma in Basic, (FOR I = 1 TO 40: POKE 1064 + I, PEEK (1064−I) : NEXT) e notate la differenza di velocità.

N.B. Nell'uno o nell'altro caso, in ragione del problema della memoria colore, per verificare il funzionamento, bisogna anche riempire di caratteri la seconda linea di schermo.

Esercizio 3.20

```
0000                       SCHERM = $22
0000                       * = $C000
C000  A9 00                LDA #0              ; PREPARA
C002  85 22                STA SCHERM          ; INDIRIZZO
C004  A9 04                LDA #4              ; SCHERMO
C006  85 23                STA SCHERM+1
C008  A2 04                LDX #4
C00A  A9 00                LDA #0              ; CODICE @
C00C  A0 00        CICLOX  LDY #0
C00E  91 22        CICLOY  STA (SCHERM),Y
C010  88                   DEY
C011  D0 FB                BNE CICLOY
C013  E6 23                INC SCHERM+1
C015  CA                   DEX
C016  D0 F4                BNE CICLOX
C018  00                   BRK
```

Esercizio 3.21

1 – La LDA = 04; STA ARR + 1 è ripetuto 4 volte. In effetti potrebbe essere incorporato all'inizio del trasferimento:

```
TRASF      LDA #$04
           STA ARR + 1
TRAS       LDX #4...
```

e si farà JRS TRASF.

Ugualmente abbiamo incorporato la LDA #$11: STA M2 in RIT all'inizio di RITAR.

2 – Potremmo fare un ciclo sulle immagini, per questo occorre un array che contenga 90, 94, 98 e 9C e si fa:

```
DISEGN     LDX #4
ANIM       LDA IMM-1,X
           STA PART + 1
           JSR TRASF
           JSR RIT
           DEX
           BNE ANIM
           JMP DISEGN
```

Il guaio è che TRASF utilizza X e che non ci sono altri registri indice (va già bene averne due). Utilizzeremo una cella di memoria al suo posto, per esempio $02.
Da cui

```
            (STA M1)
DISEGN      LDX #4
            STX CX
ANIM        LDA CX
            LDA IMM - 1,X
            STA PART + 1
            JSR TRASF
            JSR RIT
            DEC CX
            BNE ANIM
            JMP DISEGN
```

Da cui il programma completo:

```
0000                ; DISEGNO ANIMATO
0000                        PART  = $22
0000                        ARR  = $24
0000                        *  = $C000
C000  A9 04        TRANSE LDA #$04
C002  85 25               STA ARR+1
C004  A2 04        TRASF  LDX #4
C006  A0 00        CICLOX LDY #0
C008  B1 22        CICLOY LDA (PART),Y
C00A  91 24               STA (ARR),Y
C00C  88                  DEY
C00D  D0 F9               BNE CICLOY
C00F  E6 23               INC PART+1
C011  E6 25               INC ARR+1
C013  CA                  DEX
C014  D0 F0               BNE CICLOX
C016  60                  RTS
C017  A9 11        RIT    LDA #$11
C019  8D FD 03            STA M2
C01C  CE FC 03     RITAR  DEC M1
C01F  D0 FB               BNE RITAR
C021  CE FD 03            DEC M2
C024  D0 F6               BNE RITAR
C026               CX     =   $2
C026  A9 00        PP     LDA #0               ; PROGRAMMA
```

segue

seguito

```
C028  85 22                 STA PART      ; PRINCIPALE
C02A  85 24                 STA ARR
C02C  8D FC 03              STA M1
C02F  A2 04        DISEG    LDX #4
C031  86 02                 STX CX
C033  A6 02        ANIM     LDX CX
C035  BD 46 C0              LDA IMM-1,X
C038  85 23                 STA PART+1
C03A  20 00 C0              JSR TRANSE
C03D  20 17 C0              JSR RIT
C040  C6 02                 DEC CX
C042  D0 EF                 BNE ANIM
C044  4C 2F C0              JMP DISEG
C047               M1       =   $3FC
C047               M2       =   $3FD
C047  9C           IMM      .BYTE $9C
C048  98                    .BYTE $98
C049  94                    .BYTE $94
C04A  90                    .BYTE $90
```

Esercizio 3.22

Ce n'è molto poco: in C030: DISEGN LDX #6 e riallungare l'array IMM: IMM.BYTE $9C, $98, $94, $90, $C4, $C8 (più naturalmente definire le immagini).

Esercizio 3.23

Bisogna aumentare o diminuire il ritardo, quindi cambiare la costante che si mette in M2 (qui $11 all'indirizzo C018).

Per cambiare il senso di rotazione, basta cambiare l'ordine delle immagini, quindi invertire l'array IMM:

IMM.BYTE $C8, $C4, $90, $94, $98, $9C.

Esercizio 3.24

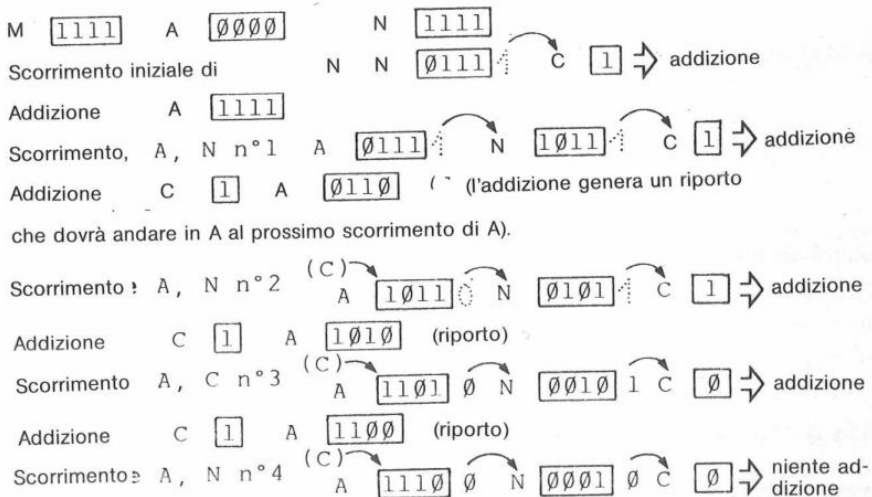
Salvataggio		Ripristino	
	PHP		PLA
	PHA		TAY
	TXA		PLA
	PHA		TAX
	TYA		PLA
	PHA		PLP

Esercizio 3.25

```
TSX
LDA $101,X
PHA
LDA $102,X
STA $101,X
PLA
STA $102,X
```

Esercizio 3.26

```
PHA
TXA
TAY
PLA
```

Esercizio 3.27

L'ultimo scorrimento di N non comporta mai somme poiché ciò che esce, è lo 0 introdotto al momento dello scorrimento iniziale.
Si ha dunque come risultato 1110 0001 = E1 esa = 225.
È proprio 15 volte 15.

Esercizio 3.28

Il riporto uscito dall'addizione non è incorporato nel risultato. Affinché il programma funzioni, occorre che si sia sicuri che l'addizione non dia mai riporto. Il programma funziona solo per numeri ≤ 7F.

Esercizio 4.3

Cursore su $ premere #$0F spazio Return.
Non dimenticate di fare Return sulla linea seguente.
M 2000-2008 dà la visualizzazione
. : 2000 AD 0F 00 A2 0F AC 0F 00
. : 2008 00 AA . . . AA
Lo 00 è rimasto in 2008 ma AC 0F 00 (LDY) è arretrato di un byte.

Esercizio 5.1

$C000 = 49152 in decimale, quindi SYS 49152.

Esercizio 5.2

Byte alto di C000 = C0 esa = 192.
Byte basso = 0 da cui
POKE 785,0: POKE 786,192: Y = USR(X).

Esercizio 5.3

Esempio di programma chiamante in Basic:
10 A = 3 : B = 5
20 TIZIO = 10 : M = 20
30 SYS 49152
40 PRINT PEEK (254) + 256 × PEEK (255).

```
0000                PTB    =   45
0000                PTH    =   PTB+1
0000                FINB   =   PTB+2
0000                FINH   =   PTB+3
0000                *      =   $C000
C000  A5 2D                LDA PTB
C002  85 FE                STA 254
```

segue

seguito

```
C004  A5 2E             LDA PTH
C006  85 FF             STA 255
C008  A0 00     CICLO   LDY #0
C00A  A9 54             LDA #'T
C00C  D1 FE             CMP (254),Y
C00E  D0 07             BNE NO
C010  C8                INY
C011  A9 4F             LDA #'O
C013  D1 FE             CMP (254),Y
C015  F0 1F             BEQ RET
C017  A5 18     NO      LDA 24
C019  18                CLC
C01A  69 07             ADC #7
C01C  85 FE             STA 254
C01E  90 02             BCC SUITE
C020  E6 FF             INC 255
C022  A5 FF     SUITE   LDA 255
C024  C5 30             CMP FINH
C026  90 E0             BCC CICLO
C028  D0 06             BNE FINE
C02A  A5 FE             LDA 254
C02C  C5 2F             CMP FINB
C02E  90 D8             BCC CICLO
C030  A9 00     FINE    LDA #0
C032  85 FE             STA 254
C034  85 FF             STA 255
C036  60        RET     RTS
```

Esercizio 5.4

Basta fare ?PEEK (56).

Esercizio 5.5

POKE55, 128: POKE 56, PEEK (56)−2.

Esercizio 5.6

Abbiamo già visto un programma analogo.

```
PART = $ 0000
ARR  = $ 7F00
*    = $ C000
```

```
C000  A2  00               LDX # 0
C002  B5  00         CICLO LDA PART, X
C004  9D  00  1F           STA ARR, X
C007  CA                   DEX
C008  D0  F8               BNE CICLO
C00A  60                   RTS
```

Esercizio 5.8

Certamente no perché il programma Basic sarà più esteso che nella precedente versione.

Esercizio 5.9

0400 − 1 + 12 × 40 = 0400 (esa) − 1 + X 480 (dec) = 0400 − 1 + 1E0 = 05DF

da cui:

```
C000 A9 05          LDA #$05
C002 85 D2          STA #$D2
C004 A9 DF          LDA #$DF
C006 85 D1          STA $D1
C008 A9 14          LDA #$14; 20 esa
C00A 85 D3          STA $D3
```

Esercizio 5.10

Programma dell'esercizio 5.9 seguito da:

```
C00C A9 14          LDA #$14
C00E A0 C0          LDY #$C0
C010 20 1E AB       JSR $AB1E
C013 60             RTS
C014 42 59 45       ; codice di bye bye
C017 42 59 45
C01A 00 00
```

Esercizio 5.11

```
LDA # < INDA      ; byte basso
LDY # > INDA      ; byte alto
JSR MACC1         ; A in ACC1
LDA # < INDB
LDY # > INDB
```

```
JSR MULTM          ; A × B in ACC1
LDA # < INDC
LDY # > INDC
JSR ADDM           ; A × B + C
LDX # < INDD
LDY # > INDD
JSR ACCIM          ; risultato in D
```

Esercizio 5.12

```
LDA # 15
STA $ B8
STA $ B9           ; AS = 15
LDA # 8
STA $BA
LDA # 2            ; 2 caratteri
STA $B7
LDA # < COM
STA $BB
LDA # > COM
STA $BC
JSR $FFC0
COM  BYT $49       ; 'I'
     BYT $30       ; '0'
```

Appendice 5

Indice dei programmi

★★★

La pratica del Commodore 64

3. linguaggio macchina e assembler del 6502

Questo libro si rivolge a due fasce di persone. Una prima fascia è formata da quegli utenti del Commodore 64 che vogliono saperne di più sul funzionamento della macchina e quindi, vogliono imparare da zero il linguaggio assembler; questo libro sarà per essi una guida utilissima che li condurrà per mano dai primi passi fino a programmi di un certo respiro scritti in linguaggio macchina.

La seconda fascia è formata da coloro che, pur conoscendo l'assembler del Commodore 64, vogliono approfondire le loro conoscenze sull'interazione tra Basic, Sistema Operativo e linguaggio macchina: questo libro sarà per essi una preziosa fonte di informazioni e consigli.

Allo scopo di facilitare la comprensione, sono stati tradotti i commenti dei programmi e anche il nome delle variabili è stato adeguato al loro significato italiano.

I.S.B.N. 88.7688.**212**.X

L. 20.000